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

D-MA-Draughts: Um sistema multiagente jogador de damas automático que atua em um ambiente de alto desempenho

LÍDIA BONONI PAIVA TOMAZ

Uberlândia - Minas Gerais
2013

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

LÍDIA BONONI PAIVA TOMAZ

D-MA-Draughts: Um sistema multiagente jogador de damas automático que atua em um ambiente de alto desempenho

Dissertação de Mestrado apresentada à Faculdade de 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.

Orientador: Prof. Dra. Rita Maria da Silva Julia

Uberlândia - Minas Gerais
2013

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

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada "**D-MA-Draughts: Um sistema multiagente jogador de damas automático que atua em um ambiente de alto desempenho**" por **Lídia Bononi Paiva Tomaz** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 24 de Maio de 2013

Orientador: _____

Prof. Dra. Rita Maria da Silva Julia
Universidade Federal de Uberlândia

Banca Examinadora:

_____

Prof. Dr. Carlos Roberto Lopes
Universidade Federal de Uberlândia

_____

Prof. Dr. Cedric Luiz de Carvalho
Universidade Federal de Goiás

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

Data: Maio de 2013

Autor:	**Lídia Bononi Paiva Tomaz**
Título:	**D-MA-Draughts: Um sistema multiagente jogador de damas automático que atua em um ambiente de alto desempenho**
Faculdade:	**Faculdade de Computação**
Grau:	**Mestrado**

Autor

Dedico esta dissertação aos meus pais, Valter e Fátima, por me concederam a vida e por estarem ao meu lado em todos os momentos.

Agradecimentos

Incialmente agradeço à Deus, pela dádiva da vida.

Agradeço aos meus pais pelo apoio infindável presente em todas as etapas de minha vida. Às minhas irmãs, Alcione e Lívia, pela amizade e companheirismo que compartilhamos.

Ao meu marido Neto pela paciência e por compreender a minha ausência durante este período. Obrigada por ser estar ao meu lado!

Em especial à Professora Rita Maria Silva Julia pela orientação, dedicação e profissionalismo com que conduziu o desenvolvimento deste trabalho. Além da amizade, gentileza e compreensão presentes durante todo este período.

Aos professores da Pós Graduação da FACOM/UFU pelo conhecimento compartilhado durante o mestrado. Ao atual secretário de pós-graduação, Erisvaldo, pela amizade e prestatividade com que sempre me atendeu.

Enfim, agradeço a todos que de certa forma estiveram envolvidos nesta etapa de minha vida e que torceram por mim para a concretização deste trabalho.

Muito Obrigada!

"Embora ninguém possa voltar atrás e fazer um novo começo...Qualquer um pode começar agora e fazer um novo fim!"

Chico Xavier

Resumo

O objetivo deste trabalho é propor um sistema de aprendizagem de damas, *D-MA-Draughts* (*Distributed Multiagent Draughts*): um sistema multiagente jogador de damas que atua em ambiente de alto desempenho. O *D-MA-Draughts* se baseia na integração e no refinamento de duas arquiteturas bem sucedidas de outros dois jogadores automáticos em damas: *MP-Draughts* e *D-VisionDraughts*. O primeiro é um sistema mutiagente que não opera em um ambiente de alto desempenho e, o segundo, é um sistema monoagente que atua em ambiente de alto desempenho. Particularmente, o *D-MA-Draughts* aprimora o *MP-Draughts* no que diz respeito à dinâmica de interação entre os agentes. Do mesmo modo, ele aumenta o desempenho do *D-VisionDraughts* com um incremento na acuidade com que representa os tabuleiros e com um incremento na quantidade de processadores do ambiente. O *D-MA-Draughts* é composto por 26 agentes, sendo um denominado IIGA (*Initial Intermediate Game Agent*), especialista em fases iniciais e intermediárias do jogo e 25 agentes especializados em fases finais. Cada um destes agentes consiste de uma rede neural multicamadas (MLP) que foi treinada, através dos Métodos das Diferenças Temporais (λ), em um ambiente de processamento distribuído a fim de alcançar um alto nível de desempenho com o mínimo de intervenção humana, distintamente do jogador automático atual campeão mundial, Chinook. A busca pelo melhor movimento é guiada pelo algoritmo de busca distribuído *Young Brothers Wait Concept* (YBWC). Os estados do tabuleiro são representados pelo mapeamento NET-FEATUREMAP, que é composto por funções que descrevem características inerentes ao próprio jogo de damas. Os agentes especialistas em fases finais de jogo foram treinados para lidar com um determinado "perfil" de tabuleiros de final de jogo (*cluster*). Tais *clusters* foram obtidos através de redes Kohonen-SOM, a partir de uma base de dados com estados de tabuleiro de final de jogo obtida em partidas reais. Depois de treinados, os agentes do *D-MA-Draughts* podem atuar em uma partida seguindo duas dinâmicas de jogo. Em ambas as dinâmicas o agente IIGA iniciará a partida. A partir daí, elas vão variar conforme descrito a seguir: na primeira, ao ser atingido um estado caracterizado como final de jogo a rede Kohonen-SOM apontará o agente (dentre os agentes de final de jogo) que representa o *cluster* cujo "perfil" mais se aproxima daquele do estado corrente do tabuleiro. Deste ponto em diante, tal agente substituirá o IIGA e conduzirá a partida até o final. Na segunda dinâmica, a cada vez que o sistema automático for executar um movimento em situações de final de jogo, a Rede Kohonen-SOM será acionada para indicar qual dos agentes de final de jogo tem "perfil" mais adequado ao estado corrente de final de jogo. Este trabalho mostra que com a arquitetura proposta o *D-MA-Draughts* conseguiu melhorar o desempenho de seus predecessores. Tais melhorias se devem ao fato de que a maior acuidade com que os estados do tabuleiro são percebidos pelos agentes possibilitou-lhe tomadas de decisão mais precisas. Em contrapartida, melhorar a representação do tabuleiro ocasiona um aumento da carga de processamento do sistema. Neste contexto, o incremento do número de processadores contornou esta situação, além de ter contribuído para aprofundar a visão futura do jogador (*look-ahead*) durante a busca, o que o tornou mais apto a escolher melhores movimentos.

Palavras chave: Sistema Jogador Multiagente Distribuído, Damas, Aprendizagem por Reforço de Máquina, Redes Neurais Artificiais, Busca Distribuída, Representação de Estados por Características, Algoritmos de Clusterização, Tabelas de Transposição

Abstract

The objective behind this study is the proposal of a Draughts learning system, *D-MA-Draughts* (Distributed Multi-agent Draughts): a multi-agent Draughts player that operates in high performance environments. *D-MA-Draughts* is based on the integration and refinement of two other successful automatic Draughts architectures: *MP-Draughts* and *D-VisionDraughts*. The first is a multi-agent system, which does not operate in a high performance environment, and the second is a mono-agent system, which operates in a high performance environment. The *D-MA-Draughts* player, in particular aided in enhancing the *MP-Draughts* player with respect to the dynamics of the interaction between the agents. Similarly, it increases the performance of *D-VisionDraughts* by augmenting the accuracy with which it represents the boards along with an increase in the number of processors pertaining to the environment. The *D-MA-Draughts* player is composed of 26 agents, with one denominated as IIGA (Initial Intermediate Game Agent), which is a specialist in initial and intermediary game states and 25 agents specializing in end game states. Each of these agents consists of a multi-layer neural network (MLP), which was trained through the use of Temporal Difference Methods (λ), in a distributed processing environment, in order to achieve a high performance level with the minimum of human intervention possible, unlike the automatic player of Draughts world champion, Chinook. The search for the best move is guided by the distributed search algorithm denominated as, Young Brothers Wait Concept (YBWC). The board states are represented by NET-FEATUREMAP mapping, which is composed of functions that describe features inherent to the game of Draughts. The specialist agents in the end game states were trained to deal with a specific type of board "profile" of end game states (cluster). These clusters were obtained through Kohonen-SOM networks from a data base with end game board states obtained from real life matches. Once trained, the *D-MA-Draughts* agents are able to operate in a match by following two dynamics of the game. In both of these dynamics the IIGA agent will start the match. From this point they will vary as follows: in the first, once the board state characterized as the end game has been reached the Kohonen-SOM network will appoint the agent (from within the end game agents), which represents the cluster whose "profile" is closest to the current board state. From here on, agents of this type will substitute the IIGA and conduct the match until the end. In the second dynamic, each time that the automatic system executes a movement in an end game situation, the Kohonen-SOM network will be activated to indicate, which of the end game agents has the most adequate "profile" for the current end game state. This study shows that by using the proposed architecture *D-MA-Draughts* managed to improve the performance achieved by its predecessors. Such improvements are due to the greater accuracy with which the board states are perceived by the agents allowing them to make more precise decisions. On the other hand, improving the board representation leads to an increase in the system's processing load. In this context, the increase in the number of processors circumvented this situation besides contributing to a deeper player future vision or *look-ahead* during the search, which made it more suitable in choosing the best moves.

Keywords: Distributed Multi-agent Player System, Draughts, Machine Reinforcement Learning, Artificial Neural Networks, Distributed Search, Representation of States through Features, Cluster Algorithms, Transposition Tables

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

APHID	**A**synchronous **P**arallel **H**ierarquical **I**terative **D**eepening
DTS	**D**ynamic **T**ree **S**plitting
IIGA	**I**nitial/**I**ntermediate **G**ame **A**gent
EGA	**E**nd **G**ame **A**gent
MLP	**M**ulti **L**ayer **P**erceptron
PVS	**P**rincipal **V**ariation **S**plitting
RNA	**R**ede **N**eural **A**rtificial
SMA	**S**istema **M**uti**A**agente
SID	**S**istema de **I**nteligência **D**istribuída
SOM	**S**elf-**O**rganizing **M**ap
TT	**T**abela de **T**ransposição
YBWC	**Y**ong **B**rothers **W**ait **C**oncept

Capítulo 1

Introdução

Desde a invenção do primeiro computador programável, tem-se buscado implementar, não somente jogos, mas também jogadores automáticos inteligentes que ofereçam desafio aos seres humanos e até os superem em competições. Em 1946, Arthur L. Samuel iniciou o projeto de um programa cujo objetivo era jogar Damas de tal forma a desafiar o campeão mundial e vencê-lo. Os resultados de sua investigação, encontrados em [6] e [7], provavelmente sejam os mais antigos no campo da Aprendizagem de Máquina que tenham conseguido atingir um nível de desempenho mestre em jogos de Damas. Em seus trabalhos Samuel não foi apenas o pioneiro de inúmeras técnicas modernas usadas em jogadores automáticos com alto desempenho, como a utilização do algoritmo poda Alfa-Beta, mas também criou um grande número de técnicas automáticas de aprendizado como o treinamento por *self-play* com clonagem, Aprendizado por Reforço com Diferenças Temporais e funções de aproximação. Samuel também notou que Damas é um bom domínio para o estudo de técnicas de aprendizado automático, uma vez que muitos dos problemas da vida real em jogos são simplificados, permitindo assim, que haja foco apenas nos problemas de aprendizado [6].

Em paralelo aos trabalhos de Samuel, a Teoria dos Jogos se tornou um importante segmento da Matemática e da Computação, especialmente após a publicação do clássico livro "The Theory of Games and Economic Behavior" por John Von Neumann e Oskar Morgenstern em 1944 [8], uma vez que averigua estratégias apropriadas para serem aplicadas em situações nas quais os resultados não dependem do comportamento de um único agente, mas também das estratégias de outros agentes que interagem com o primeiro [9].

Em [10], Herik faz uma exaustiva análise das principais características de um jogo que influenciam em sua complexidade. Basicamente duas medidas de complexidade são definidas: a complexidade do espaço de estados e o fator de ramificação da árvore de jogo. A complexidade do espaço de estados de um jogo é definido por um conjunto de todos os possíveis estados oriundos dos movimentos permitidos que possam ser executados sucessivamente no jogo a partir de seu estado inicial padrão. O fator de ramificação da árvore de jogo é definido pelo número de folhas obtidas na solução do jogo a partir da posição atual (ou estado). A principal conclusão feita por Herik é que a complexidade do espaço de estados é mais relevante do que o fator de ramificação para determinar o grau de dificuldade do jogo. A tabela 1.1, compara o fator de ramificação e o espaço de estados de alguns jogos que possuem significativa complexidade.

Tabela 1.1: Complexidade do espaço de estados e fator de ramificação de alguns jogos [5]

Jogo	**Fator de Ramificação**	**Espaço de Estados**
Xadrez	30 - 40	10^5
Damas	8 - 10	10^{17}
Gamão	+- 420	10^{20}
Othello	+-5	$< 10^{30}$
Go 19x19	+-360	10^{160}
Abalone	+- 80	$< 3^{61}$

A escolha do jogo de Damas como domínio de verificação da eficiência das técnicas da Inteligência Artificial deve-se ao fato de que ele apresenta uma complexidade significativa, conforme apresentado na tabela acima, e, por outro lado, apresenta significativas semelhanças com inúmeros problemas práticos do dia-a-dia, conforme observado por Samuel em [6]. Como exemplos destes problemas práticos é possível citar:

- *Problema de navegação em que os mapas são obtidos automaticamente 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 [11];
- *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. Thorpe e Anderson aplicaram o algoritmo SARSA (um método de Aprendizagem por Reforço) no controle de semáforos em uma pequena rede de tráfego urbano, demonstrando uma redução no tempo de espera nos semáforos em até 87% se comparado a uma estratégia aleatória de controle [12]. Mais recentemente, Wiering também projetou um sistema baseado em algoritmos de Aprendizagem por Reforço com o intuito de controlar semáforos - um modelo foi usado para estimar a dinâmica do tráfego e o método aplicado foi o de programação dinâmica, obtendo também resultados expressivos em comparação a outras estratégias de controle [13];
- *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 ELVIS (*Email Voice Interactive System*) de Walker [14], que cria um agente que aprende a escolher uma ótima estratégia de diálogo por meio de suas experiências e interações com os usuários humanos.

O tratamento automático de todos esses problemas apresenta dificuldades similares às dos problemas encontrados no domínio da implementação dos agentes jogadores a saber [15]:

- Aprender a se comportar em um ambiente onde o conhecimento adquirido é armazenado em uma função de avaliação;

- Escolher um mínimo de atributos possíveis que melhor caracterizem o domínio e que sirvam como um meio pelo qual a função de avaliação adquirirá novos conhecimentos (esta questão é de fundamental importância para se obterem agentes com alto nível de desempenho);
- Selecionar a melhor ação correspondente a um determinado estado ou configuração do ambiente onde o agente está interagindo (problema de otimização), levando em conta o fato de que, ao executar tal seleção, ele estará alterando o leque de opções de ações possíveis para os demais agentes envolvidos no ambiente;
- Adotar poderosas estratégias de aprendizagem que facilitem a geração de um agente com ótimo nível de desempenho.

A fim de explorar o vasto campo de pesquisa proporcionado pelo domínio dos jogos de Damas, a equipe em cujo contexto o presente trabalho de mestrado se aninha, propôs uma série bem sucedida de jogadores automáticos de damas inpirados na arquitetura do agente *NeuroDraughts*, de Mark Lynch [16], conforme atestam os trabalhos publicados em [15], [17], [18], [19], [20], [21], [22] e [23]. Como continuidade desta linha de pesquisa, o presente trabalho apresenta o *D-MA-Draughts*.

É importante ressaltar que atualmente existe um jogador automático campeão mundial homem/máquina, o Chinook [4], [24], [25]. Em 2007, sua equipe desenvolvedora provou que o jogo de damas estava parcialmente resolvido, ou seja, provou que qualquer jogo de Damas que comece a partir do tabuleiro inicial padrão, nenhum adversário poderia vencer o Chinook (no melhor dos casos, empataria [4]). Todavia, o Chinook não explora o contexto complexo e útil do problema do jogo de Damas como "laboratório" para as pesquisas em aprendizado por reforço. Tal fato é decorrente de que o Chinook foi inteiramente concebido com supervisão humana, desde a construção das potentes bases de dados de início e final de jogos alimentadas por especialistas humanos, até suas funções de avaliação ajustadas manualmente ao longo de décadas [18], diferentemente do sistema apresentado neste trabalho que foi desenvolvido para aprender com o mínimo de intervenção humana.

O *D-MA-Draughts* foi proposto para agregar e aprimorar as arquiteturas bem sucedidas de duas versões preliminares: *MP-Draughts* e *D-VisionDraughts*. Em comum com essas duas versões, o *D-MA-Draughts* apresenta o mesmo tipo de arquitetura baseada em Redes Neurais MLP (*Multi-Layer Perceptron*) [26] jogadoras que aprendem por reforço, através dos métodos das diferenças temporais (TD (λ)) [27], utilizando a estratégia de treino por *self-play* com clonagem [6], onde os tabuleiros são representados por um conjunto de funções (*features*) que descrevem as características do próprio jogo de damas. Tal representação é denominada mapeamento NET-FEATUREMAP [28]. A seguir, expõe-se o aprimoramento que o *D-MA-Draughts* insere em ambas as suas versões preliminares ao agregá-las.

O *MP-Draughts* é um sistema multiagente que atua em um ambiente não distribuído. Ele conta com um agente, denominado IIGA (*Initial/Intermediate Game Agent*), que atua em fases iniciais e intermediárias do jogo, e 25 agentes especializados em fases de final de jogo. Para identificar o melhor movimento, este sistema realiza uma busca na árvore de jogo através do algoritmo Alfa-Beta com Tabelas de Transposição. A representação dos estados de tabuleiro utiliza 14 características

dentre as propostas por Samuel [6] para compor o mapeamento NET-FEATUREMAP. Os agentes de final de jogo foram treinados para lidar com diferentes perfis (*clusters*) de tabuleiro de final de jogo. Tais *clusters* foram agrupados através de redes Kohonen-SOM a partir de uma base de dados composta por estados de tabuleiro de final de jogo obtidas em partidas reais disputadas por outros jogadores de Damas automáticos. Em sua dinâmica de jogo, o *MP-Draughts* atua da seguinte forma: o agente IIGA inicia o jogo e o conduz até que seja atingido um estado de tabuleiro caracterizado como de final de jogo (no máximo 12 peças). Neste momento, uma rede Kohonen-SOM verifica o agente (dentre os agentes de final de jogo) aquele cujo "perfil" mais se assemelha ao do estado corrente do tabuleiro. O agente escolhido assume o papel de EGA (*End Game Agent*) e conduz o jogo até o final, substituindo o IIGA. A proposta multiagente do *MP-Draughts* muito contribuiu para atacar os denominados problemas de *loops* de final de jogo. Este problema ocorre quando o jogador, mesmo em vantagem, não consegue pressionar seu adversário e alcançar a vitória. Ao invés disso, o agente começa uma sequência repetitiva de movimentos (*loops*) alternando-se entre posições inúteis do tabuleiro

O *D-VisionDraughts* é um sistema jogador de damas, monoagente, que atua em um ambiente distribuído. Ele foi desenvolvido para usufruir dos recursos da programação paralela a fim de beneficiar o módulo do sistema que exige mais carga de processamento: a busca. Para isso, o módulo de busca do *D-VisionDraughts* é composto pelo algoritmo de busca paralelo YBWC (*Young Brothers Wait Concept*) com Tabelas de Transposição. O YBWC é uma versão distribuída do algoritmo Alfa-Beta. O seu mapeamento NET-FEATUREMAP é composto por 12 características dentre as propostas por Samuel [6]. O ambiente de distribuição deste sistema contou com 10 processadores.

O *D-MA-Draughts* agrega ambos os sistemas anteriores do seguinte modo: é um sistema multiagente que atua em um ambiente distribuído, formado por 26 agentes especializados em fases distintas do jogo, ou seja, assim como no *MP-Draughts*, conta com um IIGA e 25 agentes de final de jogo. Tal como no *D-VisisionDraughts*, a busca pelo melhor movimento é guiada pelo algoritmo YBWC com Tabelas de Transposição. Contudo, o *D-MA-Draughts* introduz as seguintes melhorias em seus predecessores:

- Aumenta o número de características presentes em seu mapeamento NET-FEATUREMAP. O presente trabalho avalia o desempenho do *D-MA-Draughts* em duas situações: contando com 14 e 16 características. O objetivo é proporcionar ao agente uma melhor representação dos estados do jogo, permitindo-lhe uma percepção mais refinada do seu ambiente de atuação. Desta forma, o agente pode tomar decisões mais precisas, melhorando efetivamente sua participação em partidas. Observe que o *MP-Draughts* utilizou 14 características para compor seu mapeamento NET-FEATUREMAP. Por se tratar de um sistema que não atua em um ambiente de alto desempenho, esta decisão teve impacto negativo no que diz respeito ao tempo de treinamento dos seus agentes. Por outro lado, o *D-VisionDraughts* fez uso de 12 características na representação do seu mapeamento NET-FEATUREMAP. Este número é o mesmo utilizado por Lynch na construção do *NeuroDraughts* [28]. Ressalta-se

que o incremento do número de características aumenta consideravelmente a carga de processamento do sistema, uma vez que essa quantidade está diretamente relacionada com o número de neurônios da camada de entrada da rede neural MLP.

- Adiciona novos processadores no ambiente de distribuição. Mais precisamente, o *D-MA-Draughts* avalia os ganhos obtidos variando o número de processadores entre 10 e 16. Observe que o seu predecessor, *D-VisionDraughts*, fez uso de 10 processadores. Este incremento teve por objetivo contornar o problema do aumento da carga de processamento que ocorre quando o número de características para representação do tabuleiro por NET-FEATUREMAP aumenta. Além disso, esta melhoria também contribuiu para o aprofundamento da visão do jogador (*look-ahead*) sobre a partida. Desta forma, o agente consegue atingir profundidades maiores da árvore de busca do jogo, o que lhe permite uma avaliação mais apurada do estado corrente da partida sobre o qual deseja tomar uma decisão (escolher um movimento).

- Inclui uma nova dinâmica de jogo em partidas de disputa (em que o agente já completou seu ciclo de aprendizado). Além de empregar a dinâmica de jogo que o *MP-Draughts* executa, o *D-MA-Draughts* permite uma segunda dinâmica de atuação em partidas. Nesta dinâmica, assim como na dinâmica do *MP-Draughts*, o IIGA inicia a partida e a conduz até que seja atingido um estado de tabuleiro caracterizado como de final de jogo. Todavia, a partir deste momento, a cada vez que o sistema for executar um movimento, a rede Kohonen-SOM será acionada para indicar qual dos agentes de final de jogo tem o "perfil" mais adequado ao estado corrente da partida. Desta forma, haverá uma interação dentre os agentes de final de jogo até que a partida seja finalizada. O objetivo é verificar o comportamento dos agentes, uma vez que após a execução de cada movimento, o estado do tabuleiro pode se assemelhar mais ao perfil de um outro agente de final de jogo. Além disso, pretende-se também avaliar se esta nova dinâmica de jogo contribui para solucionar os problemas de *loops* de final de jogo.

Os resultados aqui obtidos comprovam as melhorias propiciadas pelo *D-MA-Draughts* no desempenho geral do sistema quando comparado às suas duas versões preliminares (*MP-Draughts* e *D-VisionDraughts*). Foram realizados testes comparando a arquitetura individual de um agente do *D-MA-Draughts*, mais precisamente, o IIGA, com o *D-VisionDraughts* a fim de verificar os ganhos obtidos com as melhorias empregadas na arquitetura dos agentes. Neste contexto, os resultados apontaram que, apenas com as melhorias incluídas no IIGA, já foi possível obter ganhos em relação ao *D-VisionDraughts* em termos de tempo de treinamento (devido ao número maior de processadores) e vitórias (devido ao *look-ahead* do jogador e melhor percepção do espaço de estados). Em relação ao sistema multiagente *D-MA-Draughts* foi realizada uma comparação entre o seu tempo de treinamento e o do sistema *MP-Draughts*. O resultado obtido indica que o tempo de treinamento do *D-MA-Draughts* foi consideravelmente menor. Ainda no sentido de avaliar a performance do sistema apresentado nesta dissertação, realizaram-se torneios envolvendo o *D-MA-Draughts* segundo suas duas dinâmicas de atuação em partidas e suas duas versões preliminares. Os resultados demonstraram que, em ambas as dinâmicas de atuação, o *D-MA-Draughts* teve um melhor desempenho no que diz respeito à quantidade de vitórias. Além disso, na sua segunda

dinâmica de atuação em partidas, foi possível constatar uma ligeira diminuição de *loops* de final de jogo.

Convém salientar que os bons resultados obtidos com as melhorias empregadas no IIGA do *D-MA-Draughts* permitiram a publicação do artigo "*Improving the Accomplishment of a Neural Network Based Agent for Draughts that Operates in a Distributed Learning Environment*" na conferência *IEEE International Conference on Information Reuse and Integration* (*IRI*) 2013 [29].

1.1 Estrutura da dissertação

Os próximos capítulos deste trabalho estão organizados conforme disposto a seguir:

Capítulo 2 Apresentação do referencial teórico que aborda as importantes técnicas utilizadas no desenvolvimento do sistema *D-MA-Draughts*.

Capítulo 3 Estado da arte em programas que utilizam técnicas de Inteligência Artificial para fazer com que um determinado agente aprenda a jogar, mais especificamente, agentes atuantes no domínio de jogos de Damas.

Capítulo 4 Apresentação da arquitetura geral do sistema *D-MA-Draughts* (*Distributed Multi-Agent Draughts*).

Capítulo 5 Detalhes do emprego da rede Kohonen-SOM no sistema *D-MA-Draughts* seguindo duas finalidades: atuar no processo de treinamento como ferramenta de *clusterização* de estados de tabuleiro caracterizados como final de jogo e processo de definição do EGA.

Capítulo 6 Técnicas e estruturas utilizadas na construção dos agentes do *D-MA-Draughts*.

Capítulo 7 Resultados experimentais obtidos com o *D-MA-Draughts*.

Capítulo 8 Conclusão e perspectivas de trabalhos futuros.

Capítulo 2

Fundamentos Teóricos

Este capítulo apresenta os principais conceitos das técnicas de Inteligência Artificial utilizadas na construção do presente trabalho.

2.1 Agentes Inteligentes

O que é um agente? Russell e Norvig [30] definem um agente como tudo que pode ser capaz de perceber seu ambiente por meio de sensores e de agir sobre esse ambiente por intermédio de atuadores. A Figura 2.1 ilustra tal ideia.

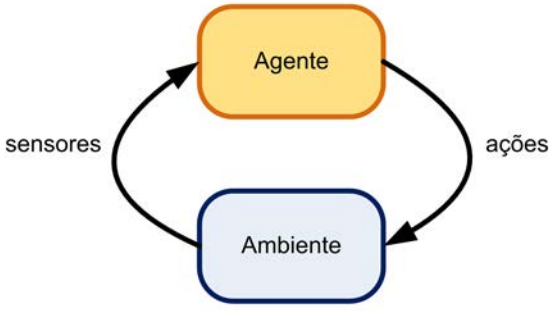

Figura 2.1: Agentes interagem com ambientes por meio de sensores e atuadores

Na Figura 2.1 é possível observar as ações sendo executadas pelo agente em função da percepção do ambiente. Esta percepção depende do controle (conhecimento) que o agente tem sobre o ambiente. Todavia, nem sempre um controle total sobre o ambiente é possível devido ao domínio do problema. Neste contexto, o agente deverá dispor de um conjunto de ações compatíveis com o ambiente que deverá modificar.

Russell and Norvig [30] sugerem as seguintes propriedades de classificação de ambientes:

- *Completamente observável versus parcialmente observável:* Um ambiente é completamente observável se o agente pode obter informações completas e precisas sobre o estado do ambiente. Em contrapartida, se o agente não consegue obter todas as informações sobre o estado do ambiente, ele é dito parcialmente observável. Por exemplo, se for considerado um robô que possui a função de limpeza de um determinado local composto por três estados A, B e C. Se este robô possui conhecimento total destes três estados, ou seja, conhece todos os obstáculos e como atuar em cada um, é dito que o agente está atuando em um ambiente completamente observável. Em contrapartida, se o agente possui conhecimento apenas do estado A, ele está inserido em um ambiente parcialmente observável, uma vez que ele não consegue perceber os estados B e C.

- *Determinístico versus Não determinístico:* Se o próximo estado do ambiente é completamente determinado pelo estado corrente e pela ação executada pelo agente ele é dito determinístico. Caso contrário, é não determinístico ou estocástico. Se o ambiente é completamente observável o agente não precisa se preocupar com a incerteza, desta forma, o agente estaria incluído em um ambiente determinístico. No entanto, se o ambiente for parcialmente observável, o agente poderia estar incluído em um ambiente não determinístico. Russell e Norvig [30] apontam que esta afirmação é verdadeira se for considerado um ambiente complexo, onde é difícil manter o controle de todos os aspectos não observados. Por exemplo, a ação de um agente dirigir um veículo como um táxi é claramente não determinístico, uma vez que não se pode prever o comportamento do tráfego com exatidão. Todavia, um robô poderia explorar um ambiente limitado onde ele teria conhecimento de todo o ambiente, desta forma ele estaria atuando em um ambiente determinístico.

- *Episódico versus Sequencial:* Em um ambiente episódico, a experiência do agente é dividida em episódios atômicos. Cada episódio consiste na percepção do agente, seguida da execução de uma única ação. Em sistemas episódicos, um episódio não pode depender de ações executadas em episódios anteriores. Por exemplo, em uma linha de montagem um agente tem que identificar peças defeituosas, logo, ele baseia sua decisão apenas na peça atual, independente das avaliações feitas nas peças anteriores. Por outro lado, um diagnóstico médico e tratamento de enfermidades é um exemplo de um episódio sequencial: um médico deve escolher entre diversas ações (exames, tratamentos, etc) sem ter certeza do estado atual do paciente. Mesmo após uma ação ter sido escolhida, não há a certeza de como ela influenciará o paciente (um tratamento pode resultar em cura em alguns casos, mas não em outros). Desta forma, uma ação efetuada a curto prazo pode ter consequências a longo prazo.

- *Estático versus Dinâmico:* Se enquanto o agente estiver deliberando o ambiente puder sofrer alterações, este é considerado dinâmico, caso contrário, é estático. Por exemplo, a ação de um agente automático estar conduzindo um veículo, como um táxi, é claramente dinâmico, uma vez que é necessário executar ações enquanto o ambiente está sendo constantemente modificado. Por outro lado, um jogo de palavras cruzadas é um ambiente estático.

- *Discreto versus contínuo:* Um ambiente é discreto se existe um número fixo e finito de ações e percepções. Um jogo de Damas é um exemplo de um ambiente discreto. Russell e Norvig [30] citam como problema de estado contínuo dirigir um táxi.

Em resumo, agentes nada mais são do que sistemas computacionais capazes de executarem ações em algum ambiente a fim de atingir um determinado objetivo. Mas esta definição não é suficiente para considerar um agente inteligente, visto que há alguns aspectos a serem observados. Um agente inteligente deve ter uma certa autonomia flexível, ou seja, deve ser capaz de perceber o meio em que se encontra e responder satisfatoriamente às mudanças que ocorrem no mesmo. Deste modo, os agentes inteligentes devem ter a capacidade de tomar iniciativas a fim de satisfazer seus objetivos. Além disso, em muitos casos devem ter aptidão para interagir com outros agentes e, possivelmente, com pessoas, a fim de tentar atingir seus objetivos [31]. Neste contexto, flexibilidade pode ser entendida como [1]:

- *Reatividade (reactivity):* agentes inteligentes são capazes de responder em tempo hábil às mudanças que ocorrem com intuito de seguir seu objetivo.
- *Pró-atividade (pro-activeness):* agentes inteligentes não efetuam apenas simples ações em resposta ao ambiente, eles são capazes de exibir um comportamento por iniciativa.
- *Habilidade social (social ability):* agentes inteligentes são capazes de interagir com outros agentes (e possivelmente humanos) a fim de atingir sua meta.

A capacidade de o agente operar com o mínimo de intervenções externas (humana e outros) define seu grau de autonomia.

2.1.1 Sistemas Multiagente

Na seção anterior foi apresentado que agentes atuam sobre um ambiente através de atuadores e sensores a fim de atingir uma meta. No entanto, há casos em que existem mais de um agente atuando sobre o mesmo ambiente, fato que define um sistema multiagente (SMA), também denominado sistema de inteligência distribuída (SID).

O ambiente de um Sistema Multiagente pode ser caracterizado como aberto ou fechado:

- *Aberto:* O agente desenvolvido pode ser empregado em sistemas de propósitos variados.
- *Fechado:* O agente desenvolvido tem um objetivo muito específico.

Em um SMA, cada agente deve atuar sobre determinada parte do problema e interagir com os demais agentes para tentar atingir um objetivo comum ou para ajudar outros agentes a alcançarem seus objetivos.

Podem-se distinguir duas classes principais de sistemas multiagentes [32]:

- *Sistemas para resolução de problemas distribuídos*: nesses sistemas os agentes envolvidos são explicitamente projetados para, de maneira cooperativa, atingirem um dado objetivo, considerando-se que todos eles são conhecidos, à priori, e supondo que todos são benevolentes, existindo, desta forma, confiança mútua em relação às suas interações;
- *Sistemas abertos:* nesses sistemas, os agentes não são necessariamente projetados para atingir um objetivo comum, podendo ingressar e sair do sistema de maneira dinâmica. Nesse caso, a dinâmica de agentes desconhecidos precisa ser levada em consideração, bem como a possível existência de comportamento não benevolente no curso das interações.

Em ambas as classes é possível observar que deve existir algum tipo de comunicação entre os agentes. Agentes se comunicam de modo a alcançarem um objetivo "ótimo" ou o objetivo da "sociedade" a qual ele pertence. Além disso, a comunicação pode permitir aos agentes *coordenarem* suas ações e comportamentos resultando em um sistema mais *coerente* [1].

A coordenação é a propriedade dos SMA´s de executar alguma atividade em um ambiente distribuído. O grau de coordenação é a extensão em que o agente evita atividades irrelevantes, *livelock* e *deadlock* e mantém as condições de segurança aplicáveis [1].

A *coerência* representa quão bem um sistema se comporta como uma unidade. Um problema para SMA´s é como eles podem manter o controle global sem realizar esta atividade de forma explícita. Neste caso, os agentes tem de ser capazes de determinar os objetivos que partilham com outros agentes a fim de evitar conflitos desnecessários, bem como compartilhar conhecimentos adquiridos [33]. A Figura 2.2 mostra um esquema de como um agente pode tentar coordenar suas atividades e comportamentos.

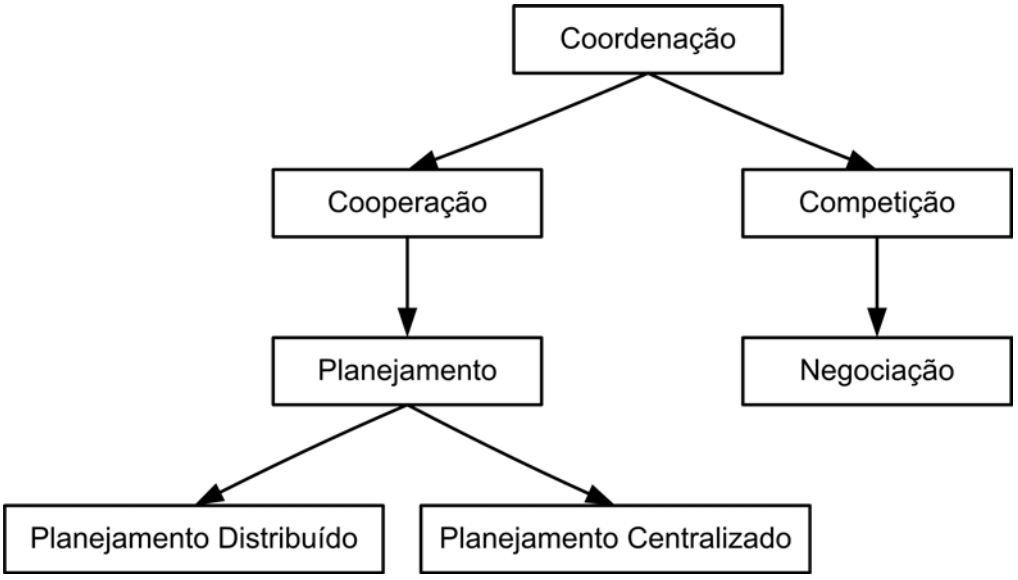

FIGURA 2.2: Uma taxonomia de algumas diferentes formas em que agentes podem coordenar seus comportamentos e atividades [1]

Russell e Norvig [30] exemplificam o comportamento de cooperação como um cenário onde um agente A (por exemplo, um motorista de táxi) interage em um ambiente com outro agente B

(por exemplo, outro veículo), Neste caso, é possível observar que se ambos agentes atuam sobre o mesmo ambiente (por exemplo, a mesma rua) seria natural que ambos os agentes cooperassem um com o outro a fim de evitar conflitos (por exemplo, uma colisão). Já o comportamento de competição poderia ser exemplificado como um cenário de jogo de Xadrez ou Damas, pois ambos os agentes tem o objetivo de ganhar a partida sobre o adversário.

Em um cenário onde existe o comportamento de *competição* também pode ser empregado o comportamento de cooperação, ou seja, dentre os diversos agentes que atuam sobre o ambiente, existem aqueles que estão ligados cooperando em prol de um objetivo, e outros que também exercem a atividade de cooperação visando outro objetivo. No caso do jogo de Damas, por exemplo, poderia haver um conjunto de agentes trabalhando (cooperando) para ganhar a partida do adversário, que por sua vez, também poderia contar com diversos outros agentes.

Moulin e Chaib-Draa [34] evidenciam as características que constituem vantagens significativas dos Sistemas Multiagente sobre um Sistema Monoagente, entre elas: maior eficiência na resolução de problemas; mais flexibilidade por possuir agentes de diferentes habilidades agrupados para resolver problemas; e aumento da segurança pela possibilidade de agentes assumirem responsabilidades de agentes que falham.

2.2 Aprendizagem por Reforço

Aprendizado por Reforço, ou simplesmente AR, é uma técnica de aprendizado de máquina onde um agente aprende por sucessivas interações em um ambiente dinâmico [35]. Ele é responsável por selecionar possíveis ações para uma determinada situação apresentada pelo ambiente [36]. Por esse motivo os agentes da AR são caracterizados como autônomos.

A questão que envolve a AR é basicamente: como um agente autônomo que atua sobre um determinado ambiente pode aprender a escolher suas ações para alcançar seus objetivos? Este é um problema muito comum em tarefas como o controle de um robô remoto e aprender a jogar jogos de tabuleiros, como Damas. O agente atua sobre o ambiente recebendo sinais (reforço ou penalidade), através de uma função de recompensa, para definir a qualidade da sequência de ações [37]. Por exemplo, em um jogo de tabuleiro como Damas, quando o agente consegue uma vitória ele receberá uma recompensa positiva (maior que zero, por exemplo), caso perca o jogo sua recompensa será negativa (menor que zero), mas se empatar sua atuação será neutralizada (recompensa igual a zero, por exemplo) [38].

A importância de se utilizar AR como 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). Um sistema típico de aprendizagem por reforço constitui-se, basicamente, de um agente interagindo em um ambiente via percepção e ação. O agente percebe as situações dadas no ambiente (pelo menos parcialmente) e seleciona uma ação a ser executada

em consequência de sua percepção. A ação executada muda, de alguma forma, o ambiente; e as mudanças são comunicadas ao agente por um sinal de reforço [15].

Formalmente, o modelo de um sistema de aprendizagem por reforço consiste em [27]:

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

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

Dentre todos os algoritmos existentes para solucionar o problema da aprendizagem por reforço, este trabalho enfocará o algoritmo de Diferenças Temporais (TD(λ)) de Sutton [27], descrito na subseção a seguir.

2.2.1 Método das Diferenças Temporais

As Diferenças Temporais são capazes de utilizar o conhecimento prévio de ambientes parcialmente conhecidos para predizer o comportamento futuro. Aprender a predizer é uma das formas mais básicas e predominantes em aprendizagem. Alguns exemplos de cenários onde alguém aprenderia a predizer seriam:

- avaliar 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 TD(λ) são guiados pelo erro ou diferença entre predições sucessivas temporárias de estados sequenciais experimentados por um agente em um domínio, resultante de uma sequência de ações $(M_0, ..., M_{i-1}, M_i, M_{i+1}, ..., M_t)$ que são executadas ao longo do tempo com o objetivo de realizar uma tarefa para o qual foi projetado. Assim, o aprendizado do agente pelo método TD(λ) é extraído de forma incremental, diretamente da experiência desse agente sobre o domínio de atuação, atualizando as estimativas a cada passo, sem a necessidade de ter que alcançar o estado final de um episódio (um episódio pode ser definido como sendo um único estado ou uma sequência de estados de um domínio) [15], [20].

O cálculo formal do reajuste dos pesos de uma rede neural, usando o método de Diferenças Temporais TD(λ) de Sutton [39] é detalhado, posteriormente, na seção 6.3.2, em que o mesmo é usado para reajuste dos pesos da rede neural jogadora do presente trabalho.

2.3 *Clusterização*

Clusterização é a divisão de dados com base na similaridade existente entre eles formando agrupamentos ou *clusters* [40]. Isso significa que os dados pertencentes a um determinado *cluster* possuem mais características em comum se comparados com os dados pertencentes a outro *cluster*.

A Figura 2.3 mostra um conjunto de dados antes da *clusterização* (Figura 2.3(a)) e maneiras diferentes de separa-los em grupos, onde cada cor corresponde a um grupo: na Figura 2.3(b) os dados são separados em dois *clusters*; na Figura 2.3(c) em três *clusters* e na Figura 2.3(d) em cinco *clusters* [41].

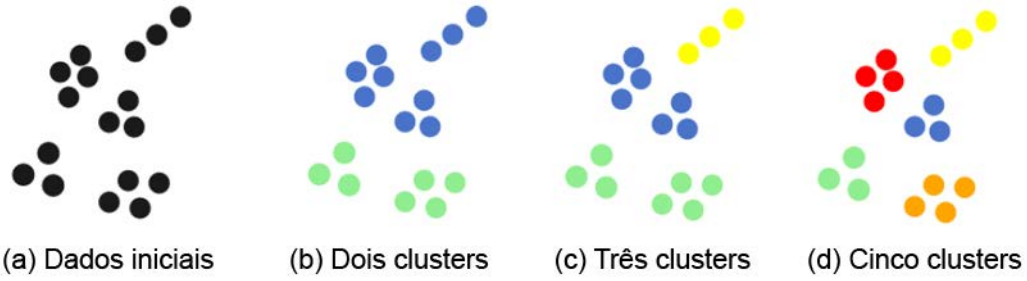

FIGURA 2.3: *Clusterização* de um conjunto de dados (a) dados originais; (b) divisão em dois *clusters*; (c) divisão em três *clusters* e (d) divisão em cinco *clusters*. Cada *cluster* é representado por uma cor diferente

É comum existir certa confusão na definição do conceito de *clusterização* definindo-o como classificação de dados. No entanto, *clusterizar* não é o mesmo que classificar dados. A principal diferença consiste no fato de que, no processo de classificação, os dados devem ser distribuídos a grupos previamente conhecidos, enquanto que no processo de *clusterização* deverá ocorrer a "descoberta" desses grupos. Por esse motivo, o ato de *clusterizar* (agrupar dados) pode ser definido como um problema de aprendizado não supervisionado, já que a estrutura dos dados e as propriedades que os tornam similares são desconhecidas [20].

Como não há rótulos iniciais, o objetivo da *clusterização* é encontrar uma organização válida e conveniente dos dados, ao invés de separa-los em categorias como acontece no reconhecimento de padrões e na classificação de dados [40].

O processo de *clusterização* é dividido em 4 etapas. Tais etapas são ilustradas na figura 2.4. Na sequência são apresentadas as etapas que compõem o processo de *clusterização*.

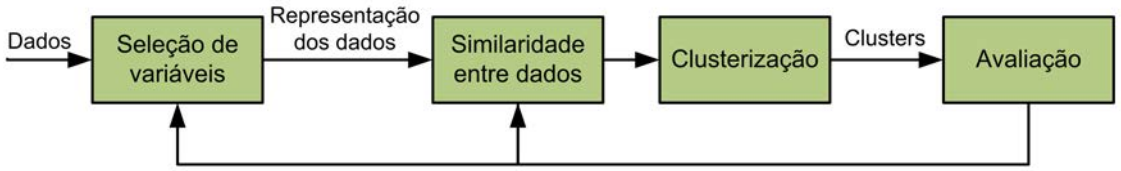

FIGURA 2.4: Etapas do processo de *clusterização*

Seleção de variáveis: esta etapa também pode ser denominada como pré-processamento, visto que consiste da identificação de variáveis ou atributos com alto grau de relevância extraídos do conjunto de dados inicial;

Medidas de similaridade entre dados: para a quantificação da proximidade de dois dados, é necessário adotar alguma medida de similaridade entre eles. Existem diversas maneiras de quantificar a similaridade, entre pares de dados, sendo que a escolha da medida de similaridade adequada é fundamental para o sucesso do processo de *clusterização* de dados. Maiores detalhes sobre medidas de similaridades serão apresentados na seção 2.3.1;

Algoritmos de clusterização: nessa etapa define-se o modo de agrupamento dos dados, que pode ser realizado de diferentes maneiras. Os algoritmos de *clusterização* classificam-se de acordo com as diferentes técnicas empregadas por eles no agrupamento dos dados. Alguns dos mais famosos algoritmos de *clusterização* serão apresentados na seção de técnicas de *clusterização* (seção 2.3.2). Nessa seção, apresentar-se-á também, brevemente, a rede Kohonem-SOM utilizada, neste trabalho, na fase de *clusterização* dos estados de tabuleiros de final de jogo;

Validação e avaliação dos resultados: nessa etapa a qualidade dos *clusters* encontrados é avaliada, já que são desconhecidos inicialmente. Essa validação pode ser feita com base em índices estatísticos ou através da comparação com outros algoritmos. Além disso, a análise dos resultados pode levar à redefinição dos atributos escolhidos e/ou da medida de similaridade, definidos nas etapas anteriores.

As quatro etapas apresentadas são essenciais para o sucesso do processo de *clusterização*, ou seja, para a obtenção de bons conjuntos de dados. Todavia, as etapas de medida de similaridade e algoritmos de *clusterização* merecem atenção especial, visto que a qualidade dos *clusters* depende muito da boa execução destas etapas. Assim sendo, estas etapas serão detalhadas nas subseções a seguir.

2.3.1 Medidas de Similaridade

Muitos métodos de *clusterização* utilizam, como ponto de partida, uma matriz que reflete, de maneira quantitativa, a proximidade entre os elementos de um conjunto de dados. Essa proximidade pode representar a distância ou similaridade entre dois elementos. Quanto maior a similaridade,

ou menor a distância entre dois elementos, mais próximos esses elementos se encontram [42]. Essa matriz recebe o nome de matriz de similaridade ou vetor de proximidade [41], [43].

As medidas de similaridade variam de acordo com a representação e escala dos atributos dos dados. Um atributo pode ter representação binária, discreta ou contínua. A escala indica o grau de importância de um atributo em relação aos demais. Por exemplo, um atributo pode indicar a porcentagem de aceitação de um produto no mercado (onde o valor do atributo varia de 0 a 100) ou indicar o peso de uma determinada característica em um tabuleiro de Damas (seção 2.10.2), onde o valor absoluto do atributo é relevante. A matriz de similaridade relaciona a proximidade entre dados do conjunto inicial.

Uma maneira de medir a similaridade entre atributos é através do cálculo da distância entre eles. A proximidade entre dois dados x_i e x_j é denotada por $d(x_i, x_j)$. A distância mais utilizada no cálculo de similaridade entre dois dados é a distância de Minkowski mostrada na Equação 2.1, onde d é o número de atributos do dado:

$$d\left(x_i, x_j\right) = \sqrt[P]{\sum_{k=1}^{d} \left(\left|x_{ik} - x_{jk}\right|\right)^2}, p \geq 1 \tag{2.1}$$

A distância de Minkowski apresenta algumas variações. A mais conhecida delas é a distância Euclidiana que é formada quando p=2, conforme Equação 2.2.

$$d\left(x_i, x_j\right) = \sqrt{\sum_{k=1}^{d} \left(\left|x_{ik} - x_{jk}\right|\right)^2} \tag{2.2}$$

Outras variações da distância de Minkowski são a distância de Manhattan, quando p=1, e a distância *sup*, quando $p \rightarrow \infty$, apresentadas nas Equações 2.3 e 2.4, respectivamente.

$$d\left(x_i, x_j\right) = \sum_{k=1}^{d} \left(\left|x_{ik} - x_{jk}\right|\right)^2 \tag{2.3}$$

$$d\left(x_i, x_j\right) = \max_{1 \leq k \leq d} \left|x_{ik} - x_{jk}\right| \tag{2.4}$$

A escolha da métrica a ser utilizada no processo de *clusterização* deve considerar os tipos de dados que serão agrupados. A variação do parâmetro p define distâncias diferentes, como pode ser observado nas Equações 2.1, 2.2, 2.3 e 2.4.

Este trabalho adota a distância Euclidiana para o cálculo de medida de proximidade entre os dados que serão agrupados (*clusterizados*), conforme será mostrado na seção 5.3

2.3.2 Técnicas de *Clusterização*

Na seção 2.3.1 relacionaram-se algumas medidas para quantificar a similaridade entre dados. O vetor de similaridade que representa essa proximidade é a entrada para os algoritmos de *clusterização*. Os algoritmos utilizados no processo de *clusterização* de dados podem ser classificados, por exemplo, de acordo com a abordagem utilizada na definição de *clusters*. Neste caso, classificam-se de acordo com 3 técnicas: particionamento; densidade e algoritmos hierárquicos; e redes auto-organizáveis. As características desses algoritmos são apresentadas, brevemente, nas subseções a seguir, dando destaque aos algoritmos de *clusterização* por redes auto-organizáveis, que foram utilizadas neste trabalho.

2.3.2.1 Algoritmos de *clusterização* por particionamento

Na *clusterização* por particionamento, o conjunto de dados é dividido em um número determinado de *clusters* uma única vez. O fato de gerar a partição uma única vez passa a ser uma vantagem, quando a quantidade de dados é muito grande e a construção e armazenamento de todas as possibilidades de divisão resultantes da *clusterização* hierárquica tornam-se muito custosas [41].

O problema de *clusterização* por particionamento pode ser definido formalmente como: dado um conjunto de n dados caracterizados por d atributos cada, determine uma partição do conjunto inicial de K *clusters*. A escolha do valor de K depende do problema abordado e pode interferir na eficiência do algoritmo. O objetivo é maximizar a similaridade entre os elementos de um mesmo *cluster* e minimizar a similaridade entre elementos de *clusters* diferentes [40].

2.3.2.2 Algoritmos de *clusterização* hierárquica

Nessa abordagem, são produzidas diversas partições do conjunto de dados com base na junção ou divisão de *clusters* de acordo com a medida de similaridade. Conforme o modo de produção das partições, os métodos de *clusterização* hierárquicos podem ser classificados em aglomerativos ou divisórios [41].

Em algoritmos aglomerativos, no passo inicial, cada elemento do banco de dados forma um *cluster*. Nas próximas iterações, pares de *clusters* da iteração precedente que satisfazem um certo critério da distância mínima são aglutinadas em um único *cluster*. O processo termina quando um número de *clusters* k fornecido pelo usuário é atingido [42].

Em algoritmos divisórios, no passo inicial, cria-se um único *cluster* composto pelo banco de dados inteiro. Nas próximas iterações, os *clusters* são subdivididos em duas partes de acordo com algum critério de similaridade. O processo termina quando se atinge um número de *clusters* k fornecido pelo usuário [42].

2.3.2.3 Redes auto-organizáveis - SOM

A rede auto-organizável desenvolvida por Teuvo Kohonen [44], denominada Kohonen-SOM (*Self Organizing Maps*), [45] pode separar dados em grupos desconhecidos inicialmente por meio de aprendizado não-supervisionado [26] [42].

Em linhas gerais, as redes auto-organizáveis são formadas por um conjunto de neurônios, onde cada dado tem seus atributos conectados a todos os neurônios da rede. A essa ligação entre neurônio e atributo é dado um peso, inicialmente, aleatório. O aprendizado ocorre à medida que os dados são apresentados à rede, e o neurônio com o conjunto de pesos mais próximo do dado é escolhido para representá-lo. O neurônio escolhido tem seus pesos alterados a fim de representar melhor o dado atribuído a ele. Assim, cada neurônio torna-se especialista na identificação dos atributos.

Redes Kohonen-SOM foi o algoritmo escolhido para o processo de *clusterização* deste trabalho. Este algoritmo foi escolhido pelo fato de sua aprendizagem ser baseada em técnicas de Inteligência Artificial (o que respeita a ideia geral deste trabalho cuja proposta é usar técnicas de Inteligência Artificial para ensinar um agente automático a jogar Damas). Outro fato importante na escolha do algoritmo de *clusterização* Kohonen-SOM foi o conhecimento que os autores deste trabalho tem sobre o mesmo [20].

Maiores detalhes sobre a rede neural Kohonen-SOM são apresentados na seção 2.4.5.

2.4 Redes Neurais Artificiais

O trabalho em redes neurais artificiais (RNA´s) 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 (sistema de processamento de informação) altamente complexo, não-linear e paralelo. Ele tem a capacidade de organizar seus neurônios de forma a realizar certos processamentos (como reconhecimento de padrões, percepção e controle motor) muito mais rápido que qualquer computador digital existente hoje [26].

Estudos realizados sobre o funcionamento do cérebro humano indicam que o mesmo desenvolve suas regras através de experiências adquiridas em situações vividas anteriormente. Tal fato serviu de inspiração para pesquisadores que tentam simular o funcionamento do cérebro, principalmente o processo de aprendizagem por "experiência", a fim de criar sistemas inteligentes capazes de realizar tarefas de classificação e reconhecimento de padrões dentre outras.

As subseções a seguir tratam os principais aspectos da abordagem de RNA´s, apresentando: a inpiração biológica para representação de um neurônio, a modelagem matemática de um neurônio e os tipos de redes neurais. Destaca-se, entretanto, que, em virtude da grande diversidade de arquiteturas encontradas na literatura, apenas as de maior importância, ou de alguma forma relevantes ao trabalho proposto serão abordados, tais como: Perceptron Multicamada e Redes Auto-Organizáveis Kohonen-SOM. Um panorama geral de outras arquiteturas pode ser encontrado em algumas referências bibliográficas da área [26], [40], [45].

2.4.1 Inspiração Biológica

O neurônio é a unidade básica do cérebro humano. Uma célula especializada na transmissão de informações, pois nelas estão introduzidas propriedades de excitabilidade e condução de mensagens nervosas [2].

Arbib mostra em [46] que a constituição de um neurônio dá-se em três partes principais: a soma ou corpo celular, do qual emanam algumas ramificações denominadas de dentritos, e por fim outra ramificação descendente da soma, porém mais extensa chamada de axônio. Nas extremidades dos axônios estão os nervos terminais, onde é realizada a transmissão das informações para outros neurônios. Esta transmissão é conhecida como sinapse. A soma e os dentritos formam a superfície de entrada do neurônio e o axônio a superfície de saída do fluxo de informação. A Figura 2.5 identifica as partes principais do neurônio e as setas mostram o fluxo da informação.

As informações que os neurônios transmitem uns aos outros são impulsos elétricos. A propagação de um estímulo ao longo dos neurônios pode ser qualquer sinal capturado pelos receptores nervosos.

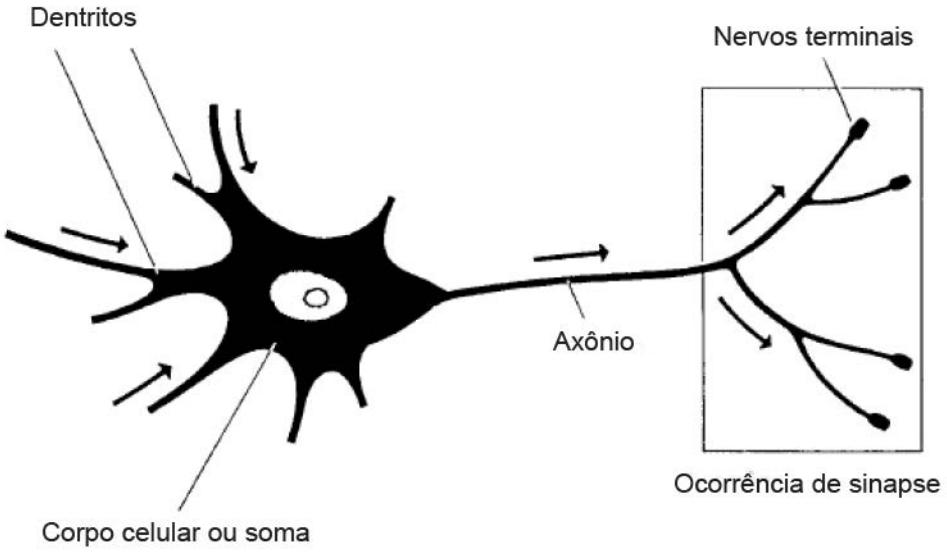

FIGURA 2.5: Modelo abstrato do neurônio biológico [2]

2.4.2 Modelo matemático de um neurônio

A partir da estrutura e funcionamento do neurônio biológico, pesquisadores tentaram simular este sistema em computadores. O primeiro modelo matemático de um neurônio artificial foi proposto por McCulloch e Pitts em 1943 [47] e está ilustrado na Figura 2.6. Neste modelo tem-se [18]:

- O neurônio é referenciado pela letra j;
- Os vínculos de entrada são representados por a_0, a_1 até a_n. Excetuando a entrada a_0 que está fixa e vale -1, as demais entradas do neurônio j representam saídas de outros neurônios de rede;

- Os pesos sinápticos são referenciados pela letra w. O peso w_{1j}, por exemplo, define o grau de importância que o vínculo de entrada a_1 possui em relação ao neurônio j;
- 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 $a_j = g(in_j) = g\left(\sum_{i=0}^{n} w_{ij}.a_i\right)$, 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.
- O peso w_{0j}, conectado à entrada fixa $a_0 = -1$, define o limite real para o neurônio j, no sentido de que o neurônio j será ativado quando a soma ponderada das entradas reais $\sum_{i=0}^{n} w_{ij}.a_i$ exceder $w_{0j}.a_0$.
- A saída do neurônio é representado por a_j.

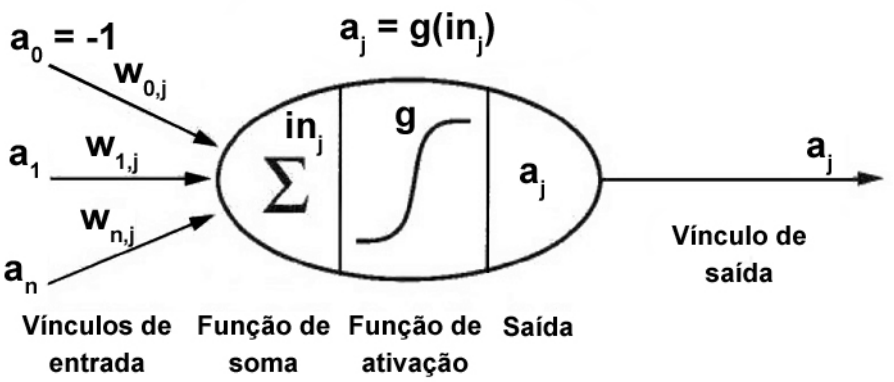

FIGURA 2.6: Modelo artificial de um neurônio proposto por McCulloch e Pitts

A função de ativação g, ou *camada de processamento de limiares*, é projetada para atender a duas aspirações [20]:

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 multicamadas, caso contrário, a rede neural se torna uma função linear simples.

Um exemplo de função de ativação é a tangente hiperbólica mostrada na Figura 2.7.

O ajuste sináptico entre os neurônios de uma RNA representa o aprendizado. Cada neurônio, conjuntamente com todos os outros, representa a informação que atravessou pela rede. Nenhum neurônio guarda em si todo o conhecimento, mas faz parte de uma malha que retém a informação graças a todos os seus neurônios. Dessa forma, o conhecimento dos neurônios e, consequentemente, da própria rede neural, reside nos pesos sinápticos.

Desse modo, pode-se dizer que as RNA´s tem sido desenvolvidas como generalizações de modelos matemáticos de cognição humana ou neurobiologia, assumindo que [15]:

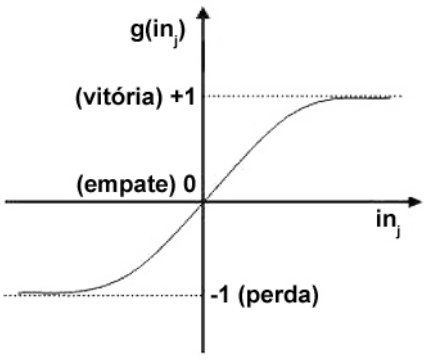

FIGURA 2.7: Modelo de função de ativação baseado na tangente hiperbólica

- 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, apresentados na próxima seção, são classificados de acordo com sua estrutura e a forma de treinamento.

2.4.3 Tipos de Redes Neurais Artificiais

A abordagem conexionista das RNAs abre um amplo leque de formas de conexão entre os neurônios (unidades de processamento). 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 RNA´s, até mesmo sua classificação gera algumas discussões. Fausett [48], por exemplo, define a 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, englobando sua forma de treinamento, finalidade, etc.

Em [49], Duc Pham define dois critérios básicos para a classificação das RNAs:

1. Quanto à estrutura;
2. Quanto à forma de treinamento.

2.4.3.1 Estrutura das Redes Neurais Artificiais

As redes neurais se classificam, quanto à estrutura em acíclicas ou cíclicas [15], [49]:

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 é o Perceptron Simples ou o Perceptron Multicamadas;

Redes cíclicas ou redes recorrentes (*recurrent*): as saídas de um ou todos os neurônios podem realimentar neurônios de camadas precedentes (tipicamente da primeira camada). Esse tipo de rede é classificada como memória dinâmica. Um exemplo típico dessa rede é a rede de Hopfield [50].

2.4.3.2 O Treinamento das Redes Neurais Artificiais

Haykin propõe a seguinte definição para o aprendizado no contexto de RNA´s: "*(...) 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*" [26].

Esta definição do processo de aprendizagem, proposta por Haykin, 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 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:

- **Redes com treinamento supervisionado**: uma sequência de padrões de entrada associados a padrões de saída é apresentada à rede; daí ela utiliza comparações entre a classificação para o padrão de entrada e a classificação correta do padrão de saída para ajustar seus pesos;
- **Redes com treinamento não-supervisionado**: não existe a apresentação de mapeamentos entrada-saída para a rede. Para este tipo de treinamento não se usa um conjunto de exemplos previamente conhecidos. Uma medida da qualidade da representação do ambiente é estabelecida e os pesos da rede são modificados de modo a otimizar tal medida;

- **Redes com aprendizagem por reforço**: utiliza-se alguma função heurística para descrever o quanto uma resposta da rede para uma determinada entrada é boa. Não se fornece à rede o mapeamento direto entrada-saída, mas sim uma recompensa (ou penalização) decorrente da saída gerada pela rede em consequência da entrada apresentada. Tal reforço é utilizado no ajuste dos pesos da rede.

O conhecimento adquirido e mantido por uma RNA reside em seus pesos, porque nenhum neurônio guarda em si todo o conhecimento, mas faz parte de uma malha que retém a informação graças a todos os seus neurônios [15]. Ajustar os pesos de uma RNA significa treiná-la com algum tipo de treinamento, seja supervisionado, não supervisionado ou com aprendizagem por reforço [20]. Particularmente, a aprendizagem por reforço é essencial para o entendimento do sistema apresentado nesta dissertação (veja seção 2.2).

2.4.4 O Perceptron Simples e Perceptron Multicamadas (MLP)

O Perceptron de uma única camada é a forma mais simples de uma RNA usada para a classificação de padrões ditos linearmente separáveis, isto é, padrões que se encontram em lados opostos de um hiperplano. Basicamente, ele consiste de uma única camada de neurônios de saída que estão conectados às entradas $x_i(n)$ pelos pesos $w_{ij}(n)$, em que $x_i(n)$ representa o i-ésimo elemento do vetor padrão de entrada na iteração n; e, $w_{ij}(n)$ representa o peso sináptico conectado a entrada $x_i(n)$ à entrada do neurônio de saída j na iteração n, como apresentado na Figura 2.8.

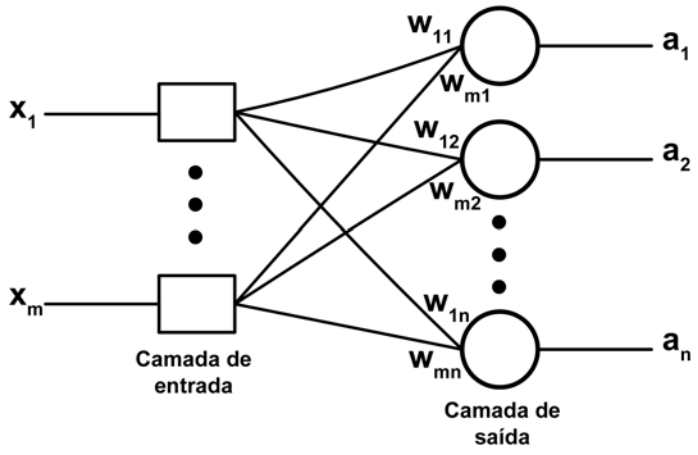

FIGURA 2.8: Arquitetura de um *Perceptron* Simples

O Perceptron de uma única camada é incapaz de resolver problemas não 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 escondidas (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 das entradas. O conjunto de saídas dos neurônios de cada camada da

rede é utilizada como entrada para a camada seguinte. A Figura 2.9 ilustra uma rede MLP com duas camadas ocultas.

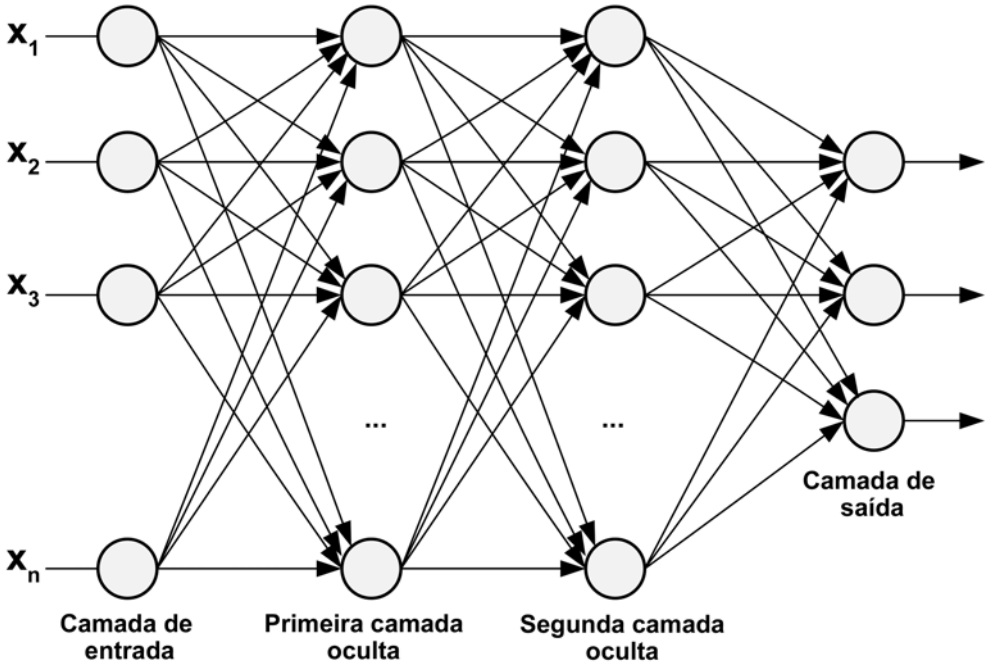

FIGURA 2.9: Arquitetura de um *Perceptron* Multicamadas - MLP

2.4.5 Redes Neurais Auto Organizáveis - SOM

Nesta seção, será estudado um dos mais populares algoritmos na categoria de aprendizado não-supervisionado, as redes neurais conhecidas como Mapas Auto-Organizados de Kohonen (Self-Organizing Map - SOM), ou simplesmente Redes Kohonen-SOM [44], [45], [51], [52].

Desenvolvidos por Teuvo Kohonen [51], estes algoritmos podem analisar dados por agrupamento com o objetivo de descobrir estruturas e padrões multidimensionais resultando na formação de classes ou *clusters*. Também faz parte de um grupo de redes neurais denominado redes baseadas em modelos de competição, ou simplesmente redes competitivas [48].

O desenvolvimento de mapas auto-organizáveis como modelo neural é motivado por uma característica do cérebro humano: o cérebro está organizado em várias regiões em que entradas sensoriais diferentes são representadas por mapas computacionais ordenados topologicamente [26]. A fim de detalhar esta característica, na próxima seção descrever-se-á a motivação biológica para a criação das redes neurais de Kohonen-SOM e, na sequência, será apresentado sua arquitetura. Estas seções restringir-se-ão a aspectos fundamentais de uma rede Kohonen-SOM.

2.4.5.1 Motivação Biológica

O córtex cerebral humano é uma fina camada de células de, aproximadamente, um metro quadrado, contendo seis camadas de neurônios redobrados para caber no crânio [53]. Apesar de a mecânica e os processos do córtex não serem ainda completamente entendidos, evidências anatômicas e

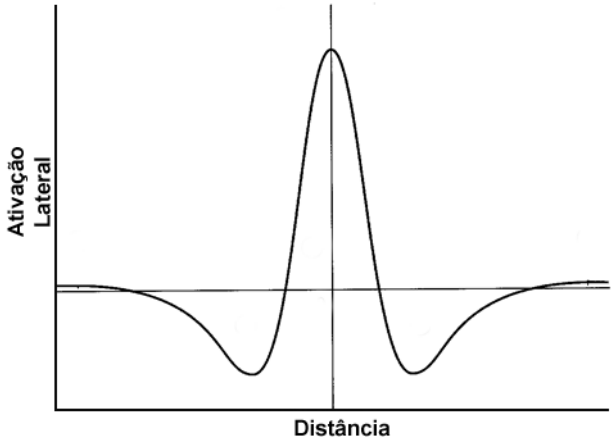

FIGURA 2.10: Função Chapéu Mexicano descrevendo a ativação lateral do cérebro humano

fisiológicas sugerem a existência de iteração lateral entre os neurônios. No caso do cérebro dos mamíferos, ao redor de um centro de excitação existe uma região (50 a 100 μm) de excitação lateral. Ao redor dessa região, existe uma área de ativação inibitória, aproximadamente, de 200 a 500 μm [45]. Novamente, ao redor dessa última área, segue-se uma região de excitação fraca (alguns centímetros). O fenômeno de iteração lateral pode ser modelado pela função chapéu-mexicano (*Mexican hat*), mostrado na Figura 2.10.

Atualmente sabe-se que o cérebro possui áreas especializadas para processar diferentes modalidades de sinais. Nessas áreas, principalmente, nas áreas dedicadas ao processamento primário de sinais sensoriais, os neurônios respondem às muitas qualidades dos estímulos de uma forma ordenada. Por exemplo, na área auditiva, existe uma "escala" para frequências acústicas diferentes. Na área visual, existem mapas para processar a orientação de segmentos de reta, mapas de cores [54] etc. Mapas ordenados topograficamente, geralmente, bidimensionais e o conceito de formação de imagens abstratas das dimensões das características sensoriais aparentam ser um dos mais importantes princípios da formação das representações internas do cérebro [45].

Talvez o trabalho experimental mais influente nessa área, tenha sido os estudos de Hubel e Wiesel [3] usando micro-eletrodos no córtex visual de gatos. Estímulos semelhantes, como segmentos de reta com orientação variando poucos graus, excitaram neurônios próximos. Registrando as respostas mais intensas com um eletrodo, inserindo paralelamente à superfície do córtex e gradualmente movido ao longo do tecido nervoso, obtém-se uma série de orientações que, em geral, variam suavemente ao longo do córtex. A Figura 2.11 mostra dados experimentais de Hubel e Wiesel, nos quais veem-se também descontinuidades ocasionais.

Percebe-se, na Figura 2.11, uma espécie de mapeamento, onde orientações similares excitam regiões do tecido nervoso próximo. No caso do córtex auditivo, experimentos indicam uma escala logarítmica de frequências. Neurônios em posições diferentes são excitados por sons diferentes e as posições relativas refletem, de uma certa forma, o relacionamento entre o conjunto de sons [45].

Apesar de as ideias de auto-organização em sistemas neurais terem sido iniciadas no final dos anos 50 e início dos anos 60 [55], C. von der Malsburg demonstrou, pela primeira vez, em 1973, a

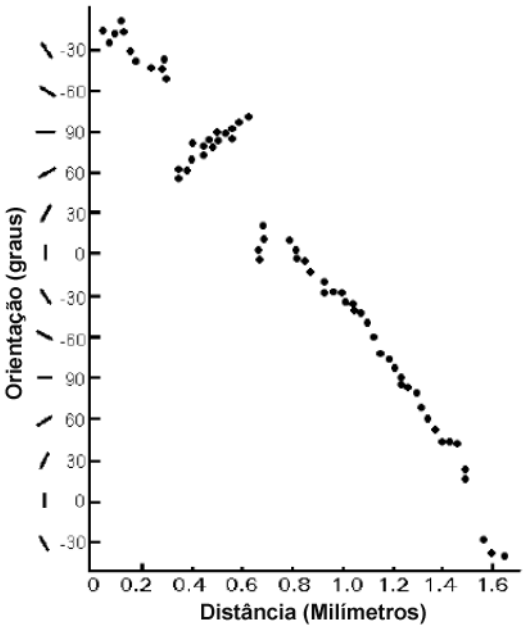

FIGURA 2.11: Sensibilidade à orientação *versus* distância - Adaptado de Hubel e Wiesel (1962) [3]

possibilidade de treinar uma rede neural usando métodos competitivos de forma a criar um mapeamento semelhante ao apresentado na Figura 2.11. Modelos de auto-organização biologicamente inspirados foram desenvolvidos [56] baseados no estudo de neurônios que respondem seletivamente a estímulos, como os sensíveis à intensidade de luz e orientação de segmentos de retas, no córtex visual.

Kohonen [44], [57] generalizou tal modelo na rede Kohonen-SOM. Previamente, Kohonen havia se dedicado a estudos sobre memória associativa e modelos para atividade neuro-biológica. A popularização da rede Kohonen-SOM deve-se a muitos fatores como, por exemplo, o seu esforço de manter uma fonte de referências sempre atualizada [20].

2.4.6 Estrutura Básica de uma Rede Kohonen-SOM

Apesar de ter sido desenvolvida inicialmente para tentar modelar áreas sensoriais do córtex e das iterações laterais entre os neurônios em tais estruturas, uma rede do tipo Kohonen-SOM tem um algoritmo extremamente simplificado e, apenas superficialmente, pode-se compará-lo a uma estrutura biológica real do cérebro.

As redes Kohonen-SOM são compostas por duas camadas, sendo a primeira a camada de entrada e a segunda de saída. A camada de entrada é por onde os dados a serem classificados (*clusterizados*) são introduzidos na rede. A segunda camada é organizada como uma grade bi-dimensional, considerada camada competitiva. Todos os nodos da camada de entrada estão interconectados com as unidades da camada de saída (camada competitiva). Em uma camada competitiva, os neurônios disputam entre si quem representará o dado apresentado na entrada de rede. Cada interconexão

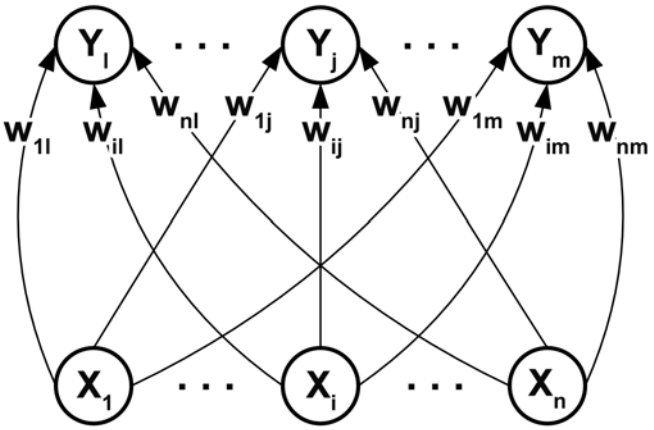

FIGURA 2.12: Modelo da arquitetura de uma rede *Kohonen-SOM*

das redes Kohonen-SOM tem um peso associado. A Figura 2.12, mostra a arquitetura deste tipo de rede.

O estado inicial da rede tem valores gerados aleatoriamente para os pesos. Tipicamente os valores de peso inicial são estabelecidos adicionando-se um número aleatório pequeno ao valor médio das entradas nos padrões de entrada. Os valores de peso são atualizados durante o treino da rede.

A arquitetura de uma rede Kohonen-SOM, mostrada na Figura 2.12, inclui n nodos na camada de entrada e m neurônios na camada de saída, onde todos os nodos X_i da camada de entrada ($1 \leq i \leq n$) estão conectados a todos os neurônios Y_j da camada de saída ($1 \leq i \leq m$). O w_{ij} indica o peso da ligação entre o i-ésimo neurônio da camada de entrada e o j-ésimo neurônio da camada de saída. Adicionalmente o vetor peso que representa cada neurônio Y_j da camada de saída é composto de todos os n pesos w_{ij} , lembrando que ($1 \leq i \leq n$).

As redes Kohonen-SOM foram utilizadas neste trabalho para atuarem em duas finalidades, conforme será detalhado no capítulo 5.

2.5 Estratégia de Busca

Os agentes inteligentes devem maximizar sua medida de desempenho para a resolução de um determinado problema. Particularmente, em um agente (jogador) que atua em jogos como Damas, o processo de busca é fundamental para a maximização do seu desempenho, visto que tal processo possibilita que o jogador encontre o melhor movimento a executar em um dado momento.

De forma genérica, as estratégias de busca tradicionais envolvem uma busca em uma árvore que descreve todos os estados possíveis a partir de um estado inicial dado. Formalmente, o espaço de busca é constituído por um conjunto de nós conectados por arcos. Cada arco pode ou não estar associado a um valor que corresponde a um custo c de transição de um nó a outro. A cada nó é associado uma profundidade p, sendo que a mesma tem valor 0 (zero) no nó raiz e aumenta de uma unidade para um nó filho. A aridade a de um nó é a quantidade de filhos que o mesmo possui, e a aridade de uma árvore é definida como a maior aridade de qualquer um de seus nós. O objetivo

da busca é encontrar um caminho (ótimo ou não) do estado inicial até um estado final explorando, sucessivamente, os nós conectados aos nós já explorados até a obtenção de uma solução para o problema. Neste contexto, a estratégia de busca pode seguir diversas abordagens sobre como os nós da árvore serão explorados. Dentre elas é possível citar a busca em profundidade limitada e a busca em aprofundamento iterativo.

2.5.1 Busca em Profundidade Limitada

Um algoritmo de busca em profundidade realiza uma busca não-informada que progride através da expansão do primeiro nó filho da árvore de busca, e se aprofunda cada vez mais, até que o alvo da busca seja encontrado ou até que ele se depare com um nó que não possui filhos (nó folha). Então a busca retrocede (*backtrack*) e começa no próximo nó. Este algoritmo possui a limitação de poder continuar insistindo em um caminho infrutífero enquanto a solução do problema pode estar em um ramo da árvore cuja distância (profundidade) ao nó raiz pode ser muito menor. Além disso, pode ocorrer da profundidade de um determinado nó ser muito longa ou até mesmo infinita. Neste caso, o algoritmo não retorna uma resposta. A fim de solucionar estes problemas, é adotada uma profundidade de busca limitada.

Na busca em profundidade limitada, também conhecida como busca de profundidade primeira, é inserido um limite máximo que pode ser atingido na exploração de um nó da árvore. Portanto, mesmo que o nó explorado ainda tenha sucessores a serem expandidos, se o limite for antigido, a busca não prossegue e retrocede a fim de iniciar a busca em outro nó [30].

2.5.2 Busca com Aprofundamento Iterativo

A qualidade de um programa jogador que utiliza um algoritmo de busca, como o Alfa-Beta (veja seção 2.6.2), depende muito do número de jogadas que ele pode olhar adiante (*look-ahead*). Para jogos com um fator de ramificação grande (observe a tabela 1.1), o jogador pode levar muito tempo para pesquisar poucos níveis adiante.

Em jogos de damas, a maioria dos sistemas jogadores utilizam mecanismos para delimitar o tempo máximo permitido de busca. Se for utilizado um algoritmo de busca em profundidade, 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, busca em profundidade não pode ser utilizada [58].

Larry Atkin [59] introduziu a técnica de aprofundamento iterativo como um mecanismo de controle do tempo de execução durante a expansão da árvore de busca. É possível notar 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 [30]. Caso o tempo se esgote e o algoritmo ainda não tenha encontrado uma solução "ótima", a busca será interrompida e o último estado analisado será retornado. A forma como esta estratégia de busca foi introduzida neste trabalho pode ser conferida na seção 6.2.4.

2.6 Algoritmos de Busca

Em problemas onde se deseja planejar, com antecedência, as ações a serem executadas por um agente em um ambiente onde outros agentes estão fazendo planos contrários àquele, surge o chamado problema de busca competitiva. Nesses ambientes, as metas dos agentes são mutuamente exclusivas. Os jogos são exemplos de ambientes que apresentam este tipo de problema de busca competitiva: o jogador não tem que se preocupar apenas em atingir seu objetivo final, mas também em evitar que algum oponente chegue antes dele, ou seja, vença o jogo. Desta maneira, o jogador deve se antecipar à jogada do seu adversário para fazer a jogada mais promissora com relação à sua meta. Esta seção apresenta o algoritmo Minimax e sua otimização através do algoritmo poda Alfa-Beta, uma vez que tais algoritmos foram propostos para solucionar este tipo de problema em jogos disputados por dois jogadores.

2.6.1 Algoritmo Minimax

O Minimax [30] é uma técnica de busca para determinar a estratégia ótima em um cenário de jogo com dois jogadores. O objetivo dessa estratégia é decidir a melhor jogada para um dado estado do jogo. Há dois jogadores no Minimax: o MAX e o MIN. Uma busca em profundidade é feita a partir de uma árvore onde a raiz é a posição corrente do jogo. As folhas dessa árvore são avaliadas pela ótica do jogador MAX, e os valores dos nós internos são atribuídos de baixo para cima com essas avaliações. As folhas do nível minimizar são preenchidas com o menor valor de todos os seus filhos, e o nível maximizar são preenchidos com o maior valor de todos os nós filhos.

A figura 2.13 mostra um exemplo de aplicação do algoritmo Minimax que gera a árvore de busca do jogo para um determinado estado.

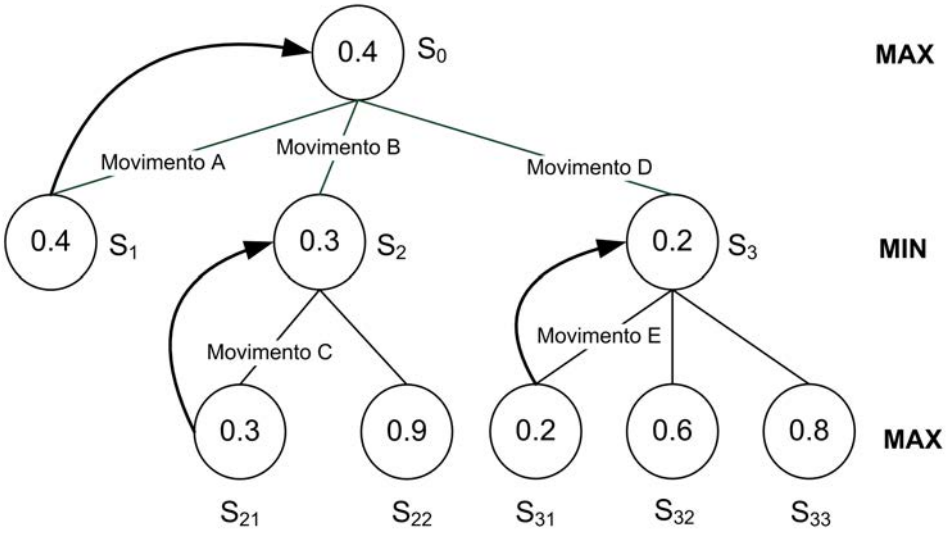

FIGURA 2.13: Árvore de busca expandida pelo algoritmo Minimax

Note que os valores das folhas 0.3 e 0.2 são retornadas para os nós S_2 e S_3, respectivamente, uma vez que estes nós estão em um nível de minimização, ou seja, recebem o menor valor de seus filhos. O valor 0.4 é retornado para S_0 que é a raiz da árvore e está em um nível de maximização (recebe

o maior valor de seus filhos). Desta forma, na Figura 2.13, o melhor movimento que o agente deve escolher é o *movimento A*.

O problema do algoritmo Minimax é que o número de estados de jogo que ele tem de examinar é exponencial em relação ao número de movimentos. Isso faz com que a busca seja demasiadamente demorada, pois diversos estados são visitados sem necessidade, ou seja, não alteram o resultado final da busca. Uma alternativa para solucionar tal problema é o emprego da técnica poda alfa-beta no algoritmo Minimax, denominado como algoritmo de busca Alfa-Beta, apresentado na seção a seguir.

2.6.2 Algoritmo Alfa-Beta

O algoritmo Alfa-Beta otimiza o algoritmo Minimax por eliminar seções da árvore que não podem conter a melhor predição [60]. Por exemplo, na Figura 2.14, o algoritmo Alfa-Beta, diferentemente do algoritmo Minimax (veja a figura 2.13) detecta que não há necessidade de avaliar as predições dos nós destacados. Portanto, o melhor movimento (*movimento A*) é encontrado mais rapidamente.

Este algoritmo recebe tal nome devido a dois valores: alfa e beta. Estes valores delimitam o intervalo que o valor da predição do melhor movimento correspondente à entrada do estado de tabuleiro deve pertencer. Desta forma, para os sucessores de um nó n, as seguintes situações podem ocorrer [30]:

- **poda** α: n é minimizador, portanto, a avaliação de seus sucessores pode ser interrompida tão logo a predição calculada para um deles (chamada *besteval*) seja menor que o valor de alfa.
- **poda** β: n é maximizador, portanto, a avaliação de seus sucessores pode ser interrompida tão logo a predição calculada para um deles (*besteval*) seja maior que beta.

O algoritmo Alfa-Beta conta com duas versões chamadas *hard-soft* e *fail-soft* que diferem apenas no valor da predição associada aos nós (ou estados de tabuleiro) dos subproblemas relacionados à raiz da árvore. Neste contexto, o valor correspondente à raíz da árvore (retorno final do algoritmo) é o mesmo valor minimax para ambas as versões [61], [62]. As subseções a seguir apresentam estas duas variantes do algoritmo Alfa-Beta.

2.6.2.1 Variante *hard-soft* do algoritmo Alfa-Beta

A variante *hard-soft* do algoritmo Alfa-Beta atua da seguinte forma, considerando que *besteval* representa o melhor valor de predição a ser associado a um nó n:

- se n é um nó maximizador, sempre que $besteval \geq beta$, o algoritmo retorna *beta* como valor mínimo da predição.

- se n é um nó minimizador, sempre que $besteval \leq alfa$, a algoritmo retorna *alfa* como valor máximo da predição.

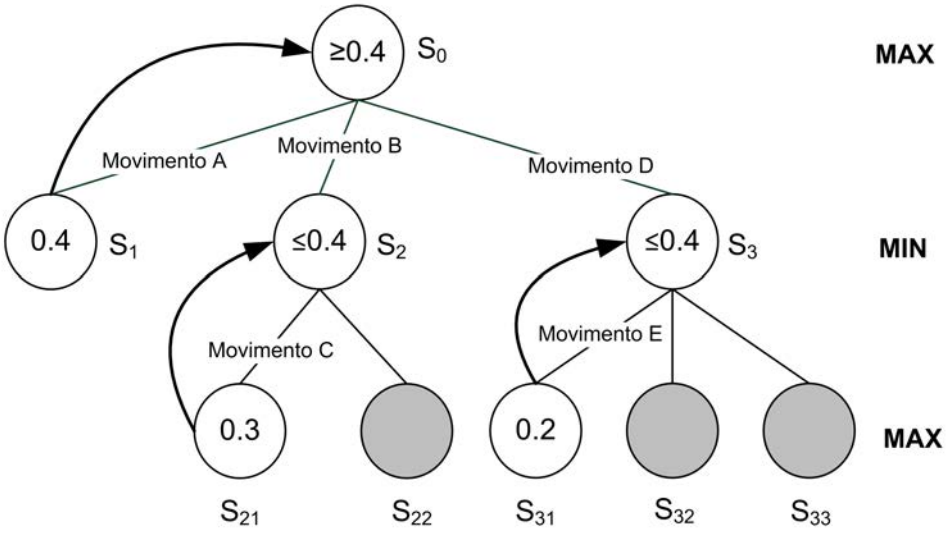

FIGURA 2.14: Árvore de busca expandida pelo algoritmo Alfa-Beta na versão hard-soft

Portanto, o valor retornado representa ou o valor minimax ou o limite imposto pelo intervalo alfa-beta. Por exemplo, na Figura 2.14, o valor ≤ 0.4 será retornado para o nó S_2 e S_3 indicando que com os movimentos B ou D, o valor da predição de ambos os nós irá ser **pelo menos** 0.4. De fato, após calcular que os valores dos sucessores mais à esquerda dos nós S_2 e S_3 tiveram predições 0.3 e 0.2, respectivamente, o algoritmo detecta que não é necessário explorar os nós destacados na figura 2.14 (que causam uma poda). Por esta razão, se eles tiverem uma predição inferior ao de seu irmão mais à esquerda, suas predições seriam candidatas a "subirem" para seus pais (que são nós minimizadores). No entanto, de qualquer forma, eles não poderiam ser escolhidos pelo nó maximizador S_0 que já tem disponível o valor 0.4 produzido pelo nó S_1. Por outro lado, se eles tiverem uma predição superior a seu irmão mais a esquerda, eles não podem ser escolhidos por seus pais (nós minimizadores).

A variante *hard-soft* possui uma limitação quando é pretendido fazer uso deste algoritmo em cojunto com uma Tabela de Transposição (TT). Uma TT é um repositório onde são armazenadas predições associadas a nós já avaliados para quando um nó for submetido ao procedimento de busca novamente (veja a seção 2.9). Por exemplo, considere novamente a Figura 2.14, se a versão *hard-soft* fosse utilizada, seria armazenado a predição 0.4 para os nós S_2 e S_3 na TT. Certamente, este valor 0.4 existente na TT poderia não ser o valor real da predição destes nós, mas dos limites impostos pelo intervalo alfa-beta. Portanto, se a variante *hard-soft* for utilizada em conjunto com uma TT para calcular o melhor movimento, e durante este processo, os estados S_2 e S_3 ocorrerem novamente, as suas respectivas predições seriam buscadas na TT. Todavia, caso estes estados encontrados na TT pudessem ser utilizados (ao cumprirem um conjunto de restrições detalhadas na seção 6.2.3.6) poderia ser realizado um movimento diferente do escolhido pelo algoritmo minimax.

Como o presente trabalho fará uso de TT para compor seu módulo de busca, uma alternativa para solucionar a limitação existente na versão *hard-soft* do algoritmo Alfa-Beta em conjunto com TT

foi adotada. Neste contexto, a próxima seção apresenta a variante *fail-soft* do algoritmo Alfa-Beta que soluciona tal limitação.

2.6.2.2 Variante *fail-soft* do algoritmo Alfa-Beta

A variante *fail-soft* do algoritmo alfa-beta atua da seguinte forma, considerando que *besteval* representa o melhor valor de predição a ser associado a um nó n:

- se n é um nó maximizador, sempre que $besteval \geq beta$, o algoritmo retorna *besteval* como valor da predição.
- se n é um nó minimizador, sempre que $besteval \leq alfa$, o algoritmo retorna *besteval* como valor da predição.

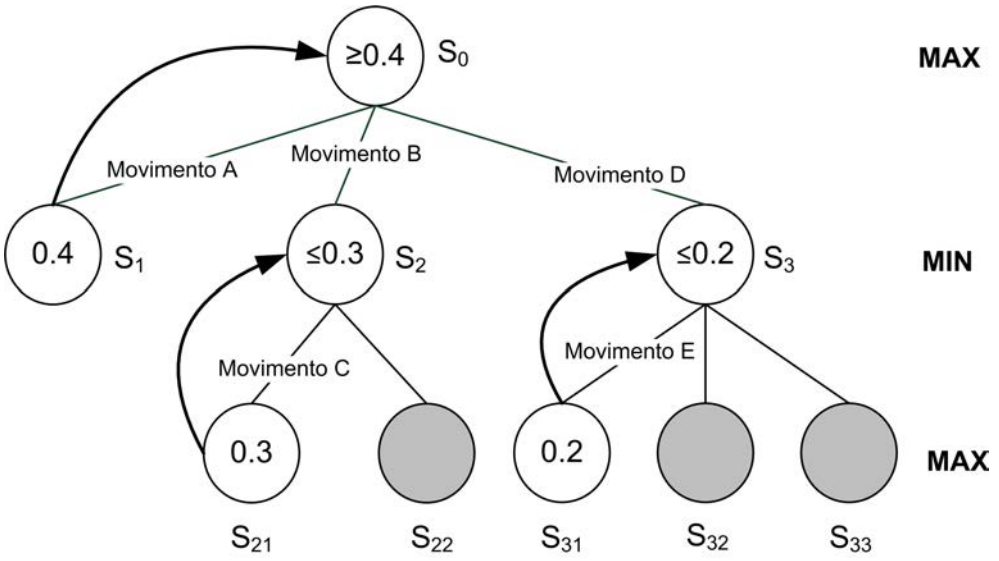

FIGURA 2.15: Árvore de busca expandida pelo algoritmo Alfa-Beta na versão fail-soft

A versão *fail-soft* é muito parecida com a versão *hard-soft*, diferindo apenas no valor das predições associadas aos nós dos subproblemas procedentes da raiz da árvore, fato que permite sua integração com TT. A versão *fail-soft* retorna o verdadeiro valor minimax para cada um dos subproblemas existentes na expansão da árvore de busca, ou seja, o valor calculado sempre representa um limite do valor minimax. Por exemplo, na Figura 2.15, quando o algoritmo observa o movimento C, ele retorna o valor 0.3 (apesar do fato de 0.3 estar fora do intervalo alfa-beta), o que significa que o movimento B possui um valor de predição igual a 0.3 [19]. Portanto, se esta versão do algoritmo Alfa-Beta for utilizada em conjunto com TT, caso um estado S_i esteja na tabela e este valor satisfaça um conjunto de restrições (detalhadas na seção 6.2.3.6 que apresenta a integração do algoritmo de busca deste trabalho com TT) o valor retornado é correspondente ao valor minimax. Desta forma, o resultado final do algoritmo não sofre alterações, ou seja, possui o mesmo retorno caso a árvore fosse expandida fazendo uso do algoritmo Minimax.

2.7 Busca Paralela

Um algoritmo paralelo pode ser executado por diferentes processadores em um mesmo espaço de tempo, retornando a mesma saída que seria gerada caso tal algoritmo fosse executado sequencialmente. O objetivo destes algoritmos é melhorar sua eficiência, visto que, normalmente, são empregados em problemas que exigem alta carga de processamento. Como exemplo, cita-se os algoritmos de busca em árvores de jogo cujo espaço de estados é elevado.

O uso de busca paralela em árvores de jogo permite realizar buscas mais profundas, afetando diretamente a visão do jogador (*look-ahead*). Jogadores com um campo de visão maior têm mais condições de fazer melhores escolhas e, consequentemente, de jogarem em um nível superior a jogadores com um campo de visão do espaço de estados menor.

Como regra, quanto mais profunda for a busca, melhor é a qualidade da jogada, pois o erro introduzido pela função de avaliação é reduzido. 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 (no xadrez este fator varia entre quatro e oito vezes) [63], [23]. Neste contexto, as subseções a seguir são dedicadas a apresentar conceitos referentes à busca paralela em árvores de jogo, que é o foco do presente trabalho. Alguns exemplos de algoritmos deste tipo de busca (com enfoque na busca Alfa-Beta paralela) serão apresentados na seção 2.8.

2.7.1 Medidas de desempenho

Uma medida de desempenho comum em algoritmos de busca paralelos em árvores 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). Pode-se definir o *speedup* S como:

$$S = T_s/T_p \tag{2.5}$$

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

O *speedup* não é normalizado para o número de processadores utilizados. Dessa forma, há um conceito derivado do *speedup* denominado eficiência. A eficiência representa quanto do potencial de *speedup* foi efetivamente alcançado, ou o quão bem o sistema foi utilizado. Em sistemas paralelos ideais, o *speedup* é menor que o 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 [23]. De um modo geral, é possível dizer que a eficiência mede a qualidade do algoritmo paralelo. A eficiência E executada em n processadores é definida como:

$$E = S/n \tag{2.6}$$

2.7.2 Dificuldades Enfrentadas na Busca Paralela

A paralelização de árvores de jogo é uma tarefa difícil [63], visto que, dentre os algoritmos desenvolvidos até o momento não há um que apresente uma eficiência ótima (ou seja, próxima ou igual a 1). Assim, é importante entender alguns problemas que cercam este tipo de busca resultando em certa ineficiência na busca paralela. Neste contexto, é possível destacar três fontes de ineficiência [64], [23]:

Sobrecarga de busca: se refere aos nós explorados pelo algoritmo paralelo que não são explorados pelo algoritmo serial. Esta sobrecarga (*overhead*) ocorre primariamente pela carência de informação pelos processadores. Em algoritmos do tipo *branch-and-bound* como o Alfa-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.

Sobrecarga de sincronização: ocorre quando um processador é forçado a esperar por informações provindas de outros processadores. Existem estratégias para reduzir tanto a sobrecarga de busca quanto a de sincronização. Porém, é um problema difícil, pois há uma estreita ligação entre as duas sobrecargas e compensações devem ser feitas com cuidado.

Sobrecarga de comunicação: É causado pela troca de informações entre os processadores e geralmente envolve o envio e recebimento de resultados entre processadores. Por conveniência, tudo o que não é atribuído às sobrecargas de busca ou sincronização é definido como sobrecarga de comunicação.

2.8 Algoritmos de Busca Paralelo baseados no Alfa-Beta

O algoritmo Alfa-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 [65]). 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 [66].

O algoritmo poda Alfa-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 sejam encontrados na literatura [66], serão apresentados apenas quatro devido a sua relevância histórica e resultados obtidos.

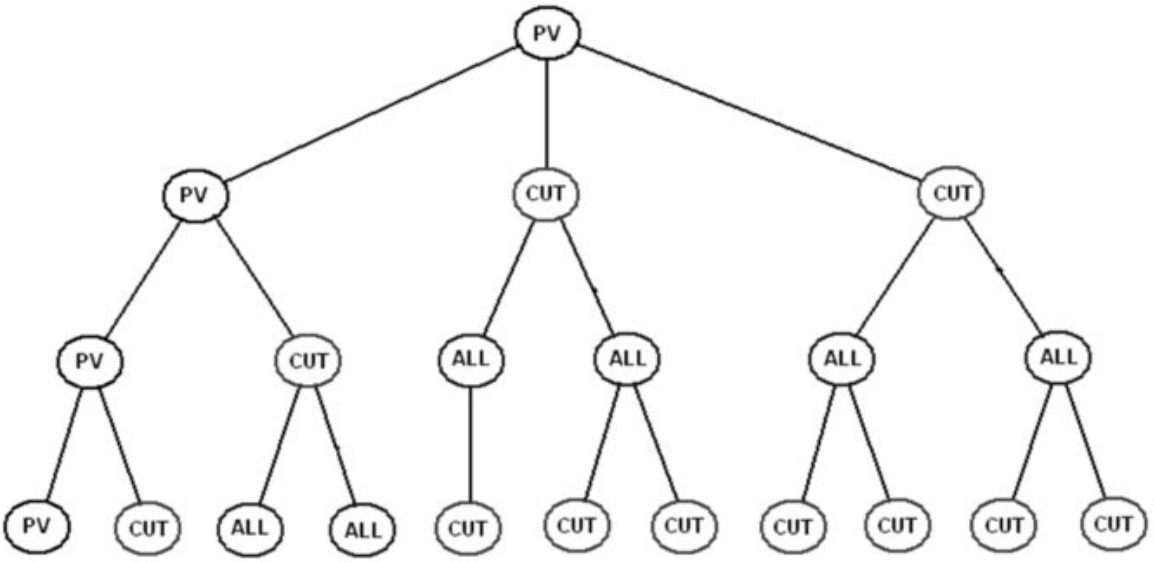

FIGURA 2.16: Tipos de nós em uma árvore de jogo.

2.8.1 PVS

O algoritmo conhecido como *Principal Variation Splitting* (PVS) foi resultado das primeiras tentativas de paralelizar o Alfa-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 [65], conforme apresentado na figura 2.16, onde:

- PV-Nodes: O nó raiz e todo primeiro sucessor de nós PV-Nodes
- CUT-Nodes: Todos os filhos de PV-Nodes, exceto o primeiro, e todos os filhos de ALL-Nodes
- ALL-Nodes: O primeiro filho de CUT-Nodes
- Undefined-Nodes: 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 na avaliação de nós PV-Nodes. Quando um PV-Node termina a avaliação de seu primeiro sucessor, todos os demais podem ser explorados em paralelo. A figura 2.17 ilustra o progresso de dois processadores, P_1 e P_2, ao explorar uma pequena árvore de busca utilizando PVS. Ambos os processadores percorrem o caminho mais à esquerda até chegarem ao nó d. Neste nó, P_2 permanece ocioso enquanto P_1 explora o ramo g. Uma vez finalizada a exploração de g, a busca paralela é iniciada: P_1 explora h enquanto P_2 explora i. Quando d tiver sido completamente avaliado, a busca é movida para o seu pai, a. Desta forma é iniciada a busca nos irmãos de d, onde o processador P_1 avalia e e o P_2 avalia f. Uma vez a tiver sido completamente avaliado, a busca paralela volta à raiz da árvore com P_1 avaliando b e P_2 avaliando c [67]. Apesar de sua importância histórica, o PVS tem uma baixa eficiência quando testado em ambientes com um grande número de processadores, mesmo com várias melhorias propostas.

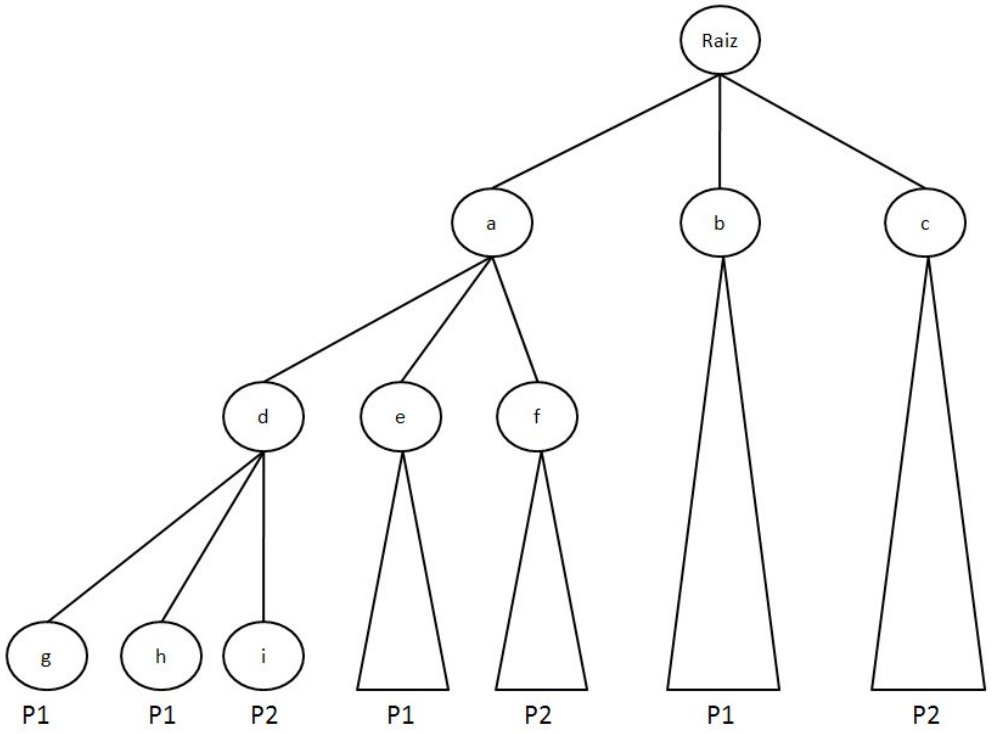

FIGURA 2.17: Exploração paralela de uma árvore com 2 processadores pelo PVS

2.8.2 YBWC

O *Young Brothers Wait Concept* (YBWC) e o *Dynamic Tree Splitting* (DTS) são propostas mais recentes e grandes *speedups* têm sido reportados [66]. Distintamente do PVS, no YBWC e no DTS, o paralelismo pode ocorrer em qualquer tipo de nós, não apenas nos nós PV-Nodes 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 apropriada 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 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 expandindo o nó mais à esquerda

da árvore ("irmão mais velho") utilizando o algoritmo Alfa-Beta como se o fizesse sequencialmente. Ao finalizar esta expansão, P_0 estará apto a dividir sua tarefa. Ao mesmo tempo, os demais 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_j 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_j responde que não tem trabalho para transferir; caso contrário, responde ao processador que faz o pedido (P_k) com o nó correspondente à raiz da sub-árvore inexplorada. A partir deste momento, P_j passa a ser o mestre de P_k e este último, o escravo, inicia uma procura sequencial seguindo as mesmas regras descritas para P_0, ou seja, expande primeiramente o ramo mais à esquerda de forma sequencial e, em seguida, estará apto a dividir sua tarefa. Os processadores podem ser mestres e escravos simultaneamente, podendo estas relações variarem dinamicamente ao longo das computações. Quando P_k termina o seu trabalho (possivelmente com o auxílio de outros processadores), ele envia uma notificação a P_j. A relação mestre-escravo é desfeita e P_k torna-se ocioso, voltando a enviar pedidos de trabalho para a rede. Este algoritmo, por ter sido adotado no presente trabalho, é descrito em maior detalhe no Capítulo 6.

2.8.3 DTS

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 o compartilhamento de informações da TT) 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 [67]. 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 do 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 com base 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 2.1.

2.8.4 APHID

O APHID (*Asynchronous Parallel Hierarchical Iterative Deepening*), proposto por Brockington [64], 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

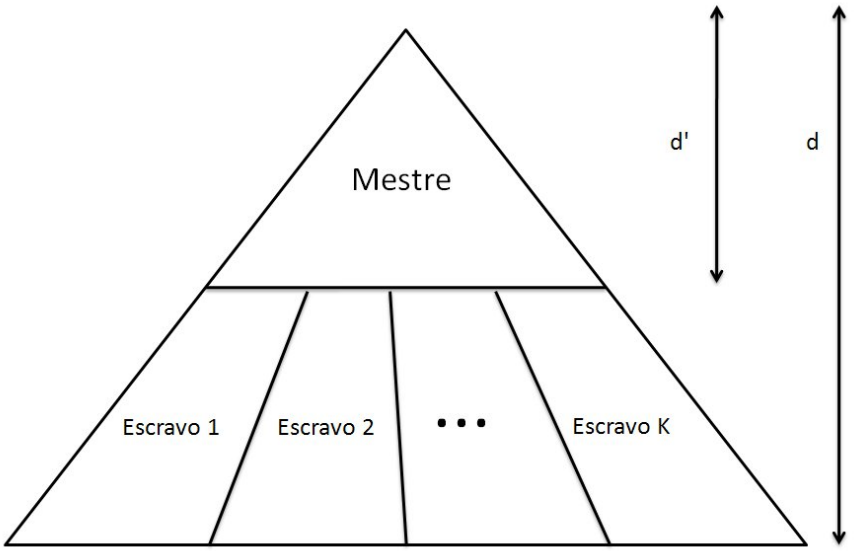

FIGURA 2.18: Exploração paralela de uma árvore pelo APHID

o trabalho é concentrado em uma parte específica da árvore ou em uma determinada profundidade de busca. Distintamente dessas abordagens, o APHID utiliza uma arquitetura mestre-escravo para realizar a busca. Um dos processadores é considerado o mestre, procura as primeiras d' jogadas e atribui estaticamente 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 as sobrecargas de comunicação e sincronização entre os vários processadores. O autor do APHID o disponibiliza através de uma biblioteca na qual, como declarado pelos autores, oferece fácil integração com jogadores existentes que usam busca serial, sendo testado com sucesso em vários jogadores inclusive no Chinook.

A tabela 2.1 resume as características dos algoritmos apresentados e introduz, na segunda coluna, tanto a hierarquia dos processadores, quanto o controle de distribuição dos algoritmos.

Tabela 2.1: Comparação algoritmos paralelos

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

2.9 Ocorrência de Transposição em Damas

Em um jogo de damas, pode-se chegar a um mesmo estado do tabuleiro várias vezes e, quando isso ocorre, diz-se que houve uma transposição [68]. As transposições ocorrem, em damas, de duas maneiras 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 2.19. Nesse caso, os estados do tabuleiro mostrados em a e d são idênticos, assim como os estados mostrados em c e f. Assumindo a como estado inicial, é possível alcançar c passando por b. Assumindo d como estado inicial, é possível alcançar f passando por e. Então, os únicos estados diferentes são b e e. No caso da sequência de movimentos a, b e c, o jogador preto move-se primeiro para a direita e, em seguida, para a esquerda, enquanto na sequência de movimentos d, e e f, o jogador preto move-se primeiro para a esquerda e, em seguida, para a direita.
- *Diferentes combinações de jogadas com reis*: os reis se movem em qualquer direção, gerando transposições facilmente, conforme mostrado na Figura 2.20. 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.

Observe que os agentes para Damas podem ter seu desempenho melhorado se contarem com um repositório para armazenar estados do tabuleiro que já tenha sido avaliados e que reapareçam por transposição. Dessa forma, eles não precisão ser reavaliados. Em virtude disso, as Tabelas de Transposição são bastante usadas para tal fim.

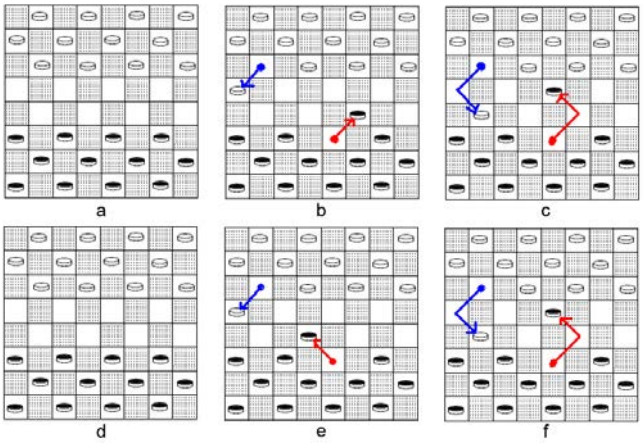

FIGURA 2.19: Exemplo de transposição em c e f: o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com peças simples

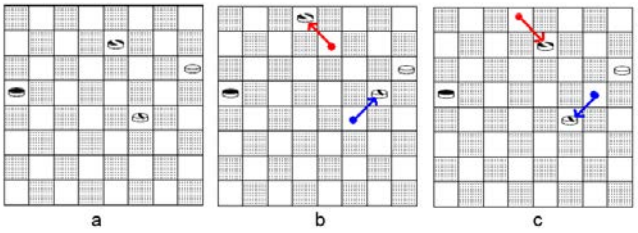

FIGURA 2.20: Exemplo de transposição em a e c: o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com reis

2.10 Representação do Tabuleiro nos Jogadores de Damas

Existem diversas maneiras de representar internamente um tabuleiro de jogo de damas [18]. Particularmente, a representação interna de um tabuleiro utilizada neste trabalho faz-se de duas maneiras: representação vetorial e representação por características, também chamada de NET-FEATUREMAP.

A representação vetorial é usada para representar o tabuleiro na entrada do algoritmo de busca durante o processo de escolha 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.

2.10.1 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).

2.10.2 Representação NET-FEATUREMAP

A utilização de um conjunto de características para treinar um jogador de damas foi primeiramente proposta por Samuel [6], 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. 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 2.2.

Conjunto de Características implementadas por Samuel

CARACTERÍSTICAS	DESCRIÇÃO FUNCIONAL
PieceAdvantage	Contagem de peças em vantagem para o jogador preto (no caso, número de peças que o jogador tem a mais).
PieceDisadvantage	Contagem de peças em desvantagem para o jogador preto (no caso, número de peças que o jogador tem a menos).
PieceThreat	Total de peças pretas que estão sob ameaça.
PieceTake	Total de peças vermelhas que estão sob ameaça de peças pretas.
Advancement	Total de peças pretas que estão na $5^{\underline{a}}$ e $6^{\underline{a}}$ linha do tabuleiro menos as peças que estão na $3^{\underline{a}}$ e $4^{\underline{a}}$ linha.
DoubleDiagonal	Total de peças pretas que estão na diagonal dupla do tabuleiro.
BackRowBridge	Se existe peças pretas nos quadrados 1 e 3 e se não existem rainhas vermelhas no tabuleiro.
CentreControl	Total de peças pretas no centro do tabuleiro.
XCentreControl	Total de quadrados no centro do tabuleiro onde tem peças vermelhas ou para onde elas possam mover.
TotalMobility - MOB	Total de quadrados vazios para onde as peças vermelhas podem mover.
Exposure	Total de peças pretas que são rodeadas por quadrados vazios em diagonal.
Taken	Total de posições para qual uma peça vermelha pode se mover e, assim, poder ameaçar uma peça preta em um movimento subsequente.
Pole	Total de peças simples pretas que estão completamente rodeada de posições vazias.
Node	Total de peças vermelhas que estão rodeada por, no mínimo, 3 posições vazias.
KingCentreControl	Total de rainhas pretas no centro do tabuleiro.
TriangleofOreo	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.
Threat	Total de posições para qual uma peça preta pode se mover e assim poder ameaçar uma peça vermelha em um movimento subsequente.
UndeniedMobility	Esta característica é a diferença entre *MOB* e *DenialofOccupancy*.
Hole	Total de posições vazias que estão rodeadas por 3 ou mais peças.

BackRowControl	Se não existem rainhas pretas e, se, uma ponte ou um triângulo de Oreo é ocupado por uma peça vermelha
Gap	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.
Exchange	Total de posições que a peça preta pode avançar uma casa e forçar o oponente a comer uma peça.
DiagonalMoment	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.
Apex	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.
Cramp	Se ocorre uma limitação de posição para as peças vermelhas e pelo menos uma das posições estão ocupadas por peças pretas.
Dyke	Total de peças vermelhas que ocupem três posições de diagonal adjacentes.
DenialofOccupancy	Total de posições onde, no próximo movimento, uma peça preta pode ocupar esta posição e ser capturada na próxima jogada.
Fork	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 pode capturar uma ou duas peças vermelhas.

Tabela 2.2: 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:

$f_i(\mathrm{x}) = a$, em que a representa o atributo do tabuleiro x referente à característica f_i, conforme exemplificado a seguir.

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 2.21 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.

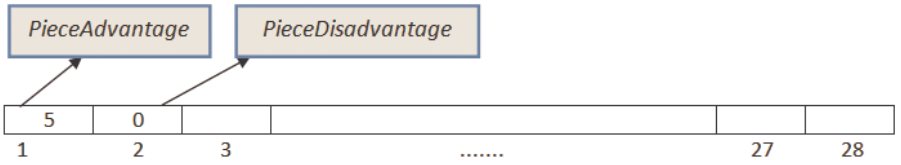

FIGURA 2.21: Exemplo de tabuleiro representado por NET-FEATUREMAP

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, que é definido pelo mapeamento C abaixo:

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

C(x) = < f_1(x),..., f_n(x) >

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 [6], [7]. 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,..., f_n, respectivamente.

Capítulo 3

Estado da Arte

Pesquisas realizadas pela comunidade científica utilizam o ambiente de jogos para aprimorar e desenvolver técnicas eficientes de Inteligência Artificial.

A criação de programas jogadores com alto nível de desempenho tem sido um dos maiores triunfos da Inteligência Artificial. A exemplo disso, tem-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. Neste contexto, a seção 3.1 apresenta alguns dos melhores sistemas jogadores de Damas automáticos já propostos que foram desenvolvidos por equipes sem relação com a equipe da linha de pesquisa a qual o *D-MA-Draughts* pertence. Posteriormente, a seção 3.2 apresenta os predecessores do *D-MA-Draughts*.

3.1 Jogadores automáticos pertencentes a outras linhas de pesquisa

Esta seção apresenta alguns jogadores automáticos para Damas que não pertencem ao grupo de pesquisa ao qual o *D-MA-Draughts* pertence.

3.1.1 Chinook

O *Chinook* foi iniciado em 1989 como uma iniciativa para tentar melhor entender as buscas heurísticas. O *Chinook* se tornou o mais famoso e forte jogador de Damas do mundo [4], [25], [24] e [69]. Ele é o campeão mundial homem-máquina para o jogo de Damas e usa uma função de avaliação ajustada manualmente para estimar quando um determinado estado do tabuleiro é favorável para o jogador. Além disso, ele tem acesso a coleções de jogadas utilizadas por grandes mestres na fase inicial de jogo (*opening books*) e um conjunto de bases de dados para as fases finais do jogo com 39 trilhões de estados do tabuleiro (todos os estados com 10 peças ou menos) com valor teórico provado (vitória, derrota ou empate). Para escolher a melhor ação a ser executada, o *Chinook* utiliza um procedimento de busca Minimax com poda Alfa-Beta, aprofundamento iterativo e TT.

O *Chinook* conseguiu o título de campeão mundial de Damas em Agosto de 1994 ao empatar 6 jogos com o Dr. Marion Tinsley que, até então, defendia seu título mundial a mais de 40 anos. Para o *Chinook*, o jogo é dividido em 4 fases e cada uma delas possui 21 características que são ajustadas manualmente para totalizar os 84 parâmetros de sua função de avaliação. Os parâmetros foram ajustados ao longo de 5 anos, por meio de testes extensivos em jogos contra si mesmo e em jogos contra os melhores jogadores humanos.

Em 2007, a equipe do *Chinook* anunciou que o jogo de Damas estava resolvido. A partir da posição inicial do jogo, existe uma prova computacional de que o jogo é um empate. A prova consiste em uma estratégia explícita com a qual o programa nunca perde, isto é, o programa pode alcançar o empate contra qualquer oponente jogando tanto com peças pretas quanto vermelhas [4].

3.1.2 Anaconda

Fogel [9], [70], [71] 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 quantidade de um estado de tabuleiro. A rede neural usou apenas posições, tipos e números 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 de 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, o Anaconda obteve 2 vitórias, 4 derrotas e 4 empates, enquanto que na segunda forma, o anaconda melhorou seu desempenho com 4 vitórias, 3 derrotas e 3 empates. Assim, o Anaconda é avaliado como um jogador especialista em damas.

3.1.3 Cake

Cake é o jogador de Damas criado por Martin Fierz [Fierz] que roda sobre a interface *Checker-Board*. A interface *Checker-Board* é a mais completa interface livre para programas jogadores de Damas da atualidade. Essa interface suporta o formato de arquivos PDN (*Portable Draughts Notation*) que é o padrão para o jogo de Damas. Com o formato PDN e a interface CheckerBoard, o jogador pode utilizar bases de dados com centenas de jogos previamente descritos, inclusive jogos de Marion Tinsley [Fierz].

Esse jogador utiliza base de dados de fim de jogo, o que significa que ele possui conhecimento perfeito do valor de qualquer jogo que tenha até 8 peças no tabuleiro. Ele também utiliza um livro de abertura de jogos (*openning book*) que contém até 2 milhões de movimentos. Entre as fases de inicio e fim de jogo, o Cake usa o algoritmo de busca MTD(f) para a escolha dos movimentos, com uma média de avaliação de 2 milhões de movimentos por segundo [Fierz].

Esse jogador já venceu uma versão anterior do programa *Chinook*, versão essa famosa pela vitória nas partidas do campeonato mundial contra Marion Tinsley (humano campeão mundial de Damas na época). Atualmente, o Cake está na versão 1.85 e, apesar de não haver grandes mudanças, é um dos agentes automáticos jogadores de Damas mais fortes que existem.

3.1.4 NeuroDraughts

O sistema *NeuroDraughts* da Lynch [28], [16] implementa o agente jogador de Damas como uma rede neural MLP (Multi-Layer Perceptron) que utiliza a busca Minimax para a escolha da melhor jogada em função do estado corrente do tabuleiro do jogo. Além disso, ele utiliza o método de aprendizagem por reforço TD(λ) aliado à estratégia de treino por *self-play* com clonagem, como ferramentas 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 (*features*), mais precisamente, o *NeuroDraughts* fez uso de 12 características escolhidas manualmente dentre as 28 propostas no trabalho de Samuel [6]. A utilização de um conjunto de características para representar o mapeamento do tabuleiro de Damas na entrada da rede neural é definida por Lynch como sendo um mapeamento NET-FEATUREMAP apresentado em detalhes na seção 2.10.2. O *NeuroDraughts* foi a base para a construção dos jogadores automáticos que compõem a linha de pesquisa que o sistema apresentado no presente trabalho se encaixa.

3.2 Jogadores automáticos predecessores do D-MA-Draughts

Nesta seção será descrito os jogadores que precederam o jogador proposto neste trabalho: *D-MA-Draughts*.

3.2.1 LS-Draughts

O *LS-Draughts* é um sistema que expande o trabalho de Lynch por meio da técnica de algoritmos genéticos, uma vez que gera automaticamente um conjunto mínimo e essencial de características para representar o jogo de Damas ao invés da definição manual utilizada pelo *NeuroDraughts*. O algoritmo genético gera T_p indivíduos que representam subconjuntos de todas as características do mapeamento NET-FEATUREMAP. Cada indivíduo gerado é anexado a uma rede neural MLP que aprende pelo método das Diferenças Temporais TD(λ). Assim como no *NeuroDraughts*, o treinamento do *LS-Draughts* utiliza a técnica *self-play* com clonagem e o mecanismo de busca adotado é o algoritmo Minimax com profundidade fixa igual a 4. O objetivo do processo de treinamento do *LS-Draughts* é detectar qual conjunto de características (*features*) é o mais conciso e apropriado para produzir um jogador eficiente.

Com o processo de escolha de características automatizado, o *LS-Draughts* supera o nível de jogo alcançado pelo *NeuroDraughts* utilizando apenas 7 características (contra 12 características utilizadas no *NeuroDraughts*): o melhor indivíduo da vigésima quinta geração derrotou o *Neuro-Draughts* com 2 vitórias e 5 empates, em um torneio com 9 jogos, conforme pode ser visto nos

resultados experimentais mostrados nos trabalhos de Neto e Julia [17], [15]. Mais detalhes sobre o *LS-Draughts* podem ser encontrados em [17], [15].

3.2.2 VisionDraughts

Com intuito de aprimorar os jogadores *NeuroDraughts* e *LS-Draughts*, Caixeta e Julia [18], [19] propuseram o *VisionDraughts*: um sistema jogador de damas automático que substitui o mecanismo de busca composto pelo algoritmo Minimax com profundidade fixa, pelo algoritmo de busca Alfa-Beta.

A arquitetura deste sistema é similar à do *NeuroDraughts*, salvo pela alteração do mecanismo de busca. Como visto em 2.5, o algoritmo Minimax examina mais estados do tabuleiro que o necessário, nesse sentido, a utilização do algoritmo Alfa-Beta elimina seções da árvore de busca que, definitivamente, não podem conter a melhor ação a ser executada pelo agente jogador fazendo com que sua tomada de decisão seja mais ágil. Ainda no sentido de melhorar o mecanismo de busca do *VisionDraughts*, foi adicionada a estratégia de busca por aprofundamento iterativo combinado com TT [18].

Os resultados do *VisionDraughts* foram bastante positivos comparados com o *LS-Draughts*: reduziu de 94,17% o tempo de busca e, com tabelas de transposição esta redução chegou a 97,73%. Além disso, em um torneio de 14 jogos contou com 4 vitórias, 8 empates e 2 derrotas [18], [19].

3.2.3 MP-Draughts

O *MP-Draughts* (*Multi Player Draughts*) é um sistema multiagente jogador de damas composto por 26 agentes especialistas em fases distintas do jogo, isto é, 1 agente denominado IIGA (*Initial/Intermediate Game Agent*) e 25 agentes de final de jogo. O IIGA corresponde ao *VisionDraughts*. Os demais agentes seguem uma arquitetura semelhante a este sistema, todavia o mapeamento NET-FEATUREMAP é composto por 14 características. O número maior de características visou contribuir no controle do problema de *loops* de final de jogo.

O IIGA é responsável por iniciar e conduzir a partida até que seja atingido um estado de final de jogo, mais precisamente, um estado de tabuleiro com no máximo 12 peças. Os agentes de final de jogo, foram criados para lidar com determinados agrupamentos (*clusters*) de estados de tabuleiro de final de jogo. No total são 25 *clusters* que foram minerados de uma base de dados (BD) composta por 4000 estados de tabuleiro de final de jogo obtidos em partidas reais do jogo de Damas. A mineração destes estados de tabuleiro foi realizada através de uma rede Kohonen-SOM (2.4.5) cuja medida de similaridade adotada foi a Distância Euclidiana. Logo, a dinâmica de jogo deste sistema ocorre da seguinte forma: o IIGA atua na partida enquanto o estado do tabuleiro tiver mais de 12 peças, tão logo seja atingido um estado contendo 12 peças ou menos um dos agentes de final de jogo irá conduzir a partida até o final, a partir deste momento, tal agente passa a ser denominado EGA (*End Game Agent*).

O *MP-Draughts* obteve resultados bem satisfatórios. Apesar do maior tempo de treinamento devido ao número de agentes a serem treinados, em dois torneios de 20 jogos contra o *VisionDraughts*, obteve 65% de vitórias [20], [21]. Em relação aos *loops* de final de jogo, foi constatado uma diminuição considerável, onde nestes mesmos torneios contra o *VisionDraughts* foi identificado apenas 5% de empates ocasionados por *loops*.

3.2.4 D-VisionDraughts

Em [22], [23] os autores propuseram o *D-VisionDraughts* (*Distributed VisionDraughts*). Este sistema é a versão distribuída do *VisionDraughts*, uma vez que substitui o algoritmo de busca sequencial Alfa-Beta pelo algoritmo de busca paralelo YBWC (*Young Brothers Wait Concept*) [66]. Além disso, o *D-VisionDraughts* trabalha com a ordenação da árvore de jogo através do uso de heurísticas.

A paralelização do algoritmo de busca permitiu ao jogador uma visão futura melhor (*look-ahead*) do estado corrente do jogo para a escolha do melhor movimento, bem como uma redução significativa do tempo de treinamento em relação a seu predecessor *VisionDraughts*.

A ordenação da árvore de busca 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. Desta forma, pode haver uma expressiva redução no número de nós avaliados, fato que aumenta a eficiência do algoritmo de busca, uma vez que permite que a busca seja concluída mais rapidamente [22], [23].

Para a ordenação da árvore de busca o *D-VisionDraughts* faz uso de uma TT e de *killer* moves [72] 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 pelo jogador. 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 TT, o movimento sugerido pela TT é colocado na primeira posição da lista de sucessores de n;
- Se o registro não foi encontrado na TT, 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.

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 permitiu ao *D-VisionDraughts*, nas mesmas condições de treino, gerar um jogador com um nível de jogo superior. De fato, em torneios contra o *VisionDraughts*, o tempo de execução do *D-VisionDraughts* (rodando em 10 processadores) foi 83% inferior ao do seu oponente e a sua porcentagem de vitória foi 15% superior [22], [23]. O "speed-up" ou desempenho do *D-VisionDraughts* em função do

número de processares utilizados para executá-lo foi avaliado através de testes experimentais e os resultados obtidos podem ser conferidos em [23].

Capítulo 4

O Sistema D-MA-Draughts

O *D-MA-Draughts* (*Distributed Multi-Agent Draughts*) é fruto de uma contínua atividade de pesquisa que teve como frutos anteriores os jogadores *LS-Draughts* [17], *VisionDraughts* [19], *MP-Draugths* [21] e *D-VisionDraughts* [22]. O objetivo dos autores destes jogadores é explorar o contexto da Aprendizagem de Máquina atuando no domínio de jogos de Damas. Tal vertente gerou como produtos agentes com desempenho comprovadamente bom e crescente, mantendo, sempre, a estratégia de aprendizado praticamente desvinculado da interferência de especialistas humanos. Desta forma, como o presente trabalho é uma extensão desta pesquisa, o *D-MA-Draughts* foi proposto e implementado de forma a atingir um bom nível de eficiência considerando o elevado espaço de estados existente no seu ambiente de atuação, conforme apresentado na introdução deste trabalho (capítulo 1). Em síntese, o *D-MA-Draughts* unifica as idéias de duas arquiteturas bem sucedidas: *MP-Draughts*, um sistema multiagente que não opera em um ambiente de alto desempenho (veja a seção 3.2.3), e *D-VisionDraughts*, um sistema monoagente que opera em um ambiente de alto desempenho (veja a seção 3.2.4). O objetivo desta unificação é obter um sistema multiagente que opera em um ambiente de alto desempenho resultando em um jogador mais robusto e que tenha mais chances de caminhar rumo a vitórias.

O bom desempenho do agente jogador está relacionado com a forma como os estados de tabuleiro são representados e com a quantidade de estados que este consegue explorar durante o processo de busca pelo melhor movimento. Este processo é feito por uma busca em profundidade na árvore de jogo, cuja raiz é formada pelo estado corrente do tabuleiro do jogo e as folhas são os estados que devem ser representados para avaliação a fim de permitir ao algoritmo de busca a identificação da melhor ação a ser efetuada. Quanto mais profunda for a exploração desta árvore de jogo, mais estados o agente terá visitado, desta forma, diz-se que a visão futura (*look-ahead*) do jogador foi beneficiada. Além disso, a representação dos estados de tabuleiro deve ser feita de maneira adequada. Todavia, para que estes requisitos possam integrar um agente jogador é exigida uma capacidade de processamento elevada. Por isso, a fim de permitir ao *D-MA-Draughts* um ambiente apropriado para explorar um número maior de estados (beneficiando seu *look-ahead*) e uma representação de estados de tabuleiro adequada, ele foi implementado em um ambiente de alta performance. Além disso, o *D-MA-Draughts* conta com uma arquitetura multiagente constituída por 26 agentes especializados em fases distintas do jogo, o que lhe permite ter uma

visão mais apurada da partida. Um agente, denominado IIGA (*Initial/Intermediate Game Agent*), é especializado em fases iniciais e intermediárias da partida. Os demais agentes, denominados agentes de final de jogo, foram construídos e treinados para serem especialistas em fases de final de jogo. Em damas, o jogo começa a assumir aspectos de desfecho de partida quando o tabuleiro conta com cerca de 12 peças. Assim sendo, o IIGA opera enquanto o tabuleiro tem no máximo 13 peças. A partir daí, os agentes de final de jogo entram em cena assumindo a partida. Conforme será visto na seção 5.3.2 o desempenho do *D-MA-Draughts* será avaliado segundo duas dinâmicas distintas de atuação dos agentes de final de jogo. Estas dinâmicas visam contribuir na resolução dos problemas de *loops* de final de jogo. Um *loop* ocorre quando o agente, mesmo em vantagem, não consegue pressionar o adversário e alcançar a vitória. Ao invés disso, o agente começa uma sequência repetitiva de movimentos (*loop*) alternando-se entre posições inúteis do tabuleiro [20].

A seguir é apresentada a arquitetura multiagente do sistema *D-MA-Draughts*.

4.1 Arquitetura Multiagente do D-MA-Draughts

Esta seção dedica-se a apresentar a arquitetura multiagente do sistema *D-MA-Draughts*, que é similar à do seu precursor *MP-Draughts*. Tal arquitetura é ilustrada na Figura 4.1.

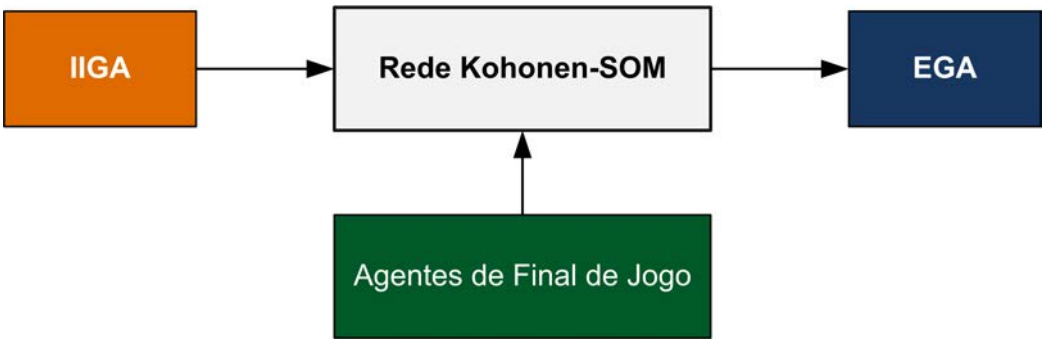

FIGURA 4.1: Arquitetura do jogador *D-MA-Draughts*

Conforme a figura, os módulos do *D-MA-Draughts* são os seguintes:

IIGA - *Initial/Intermediate Game Agent*: esse agente é treinado para ser especialista em fases iniciais e intermediárias de jogo, o que aqui lhe acarreta a responsabilidade de conduzir a partida em tabuleiros com, no mínimo, 13 peças. O agente IIGA é o refinamento do agente *D-VisionDraughts* proposto por Barcelos e Julia [22]. Os detalhes técnicos e estruturais deste agente serão apresentados no capítulo 6.

Redes Kohonen-SOM : o referido módulo possui duas finalidades:

1. *Clusterização de estados de tabuleiro de final de jogo segundo suas similaridades*: aqui a rede Kohonen-SOM é treinada para ser responsável por agrupar milhares de tabuleiros de final de jogo presentes em uma base de dados obtidos em jogos reais. Após treinada, a rede Kohonen-SOM deve ser capaz de apontar a qual dos grupos gerados pertence

um determinado estado de tabuleiro de final de jogo, baseando-se, para tanto, na similaridade deste tabuleiro com cada grupo (ele pertencerá àquele do qual for mais próximo). Conforme será visto na seção 5.2.1, para a base de dados adotada o número de *clusters* obtido foi de 25. Para cada *cluster* do *D-MA-Draughts* é associada uma rede MLP que será treinada de forma a se especializar no *perfil* de tabuleiro de final de jogo do *cluster* que ela representa. Tais MLP´s consistem dos agentes de final de jogo. Assim sendo, haverá 25 desses agentes.

2. *Classificação do EGA dentre os agentes de final de jogo* - neste caso, a rede deverá definir, dentre os 25 agentes de final de jogo o mais apto a conduzir a partida quando esta atingir um estado de final de jogo (EGA).

Os detalhes de cada uma destas atuações da rede Kohonen-SOM no *D-MA-Draughts* serão apresentados no Capítulo 5.

Agentes de Final de Jogo : conforme dito acima, esse módulo consiste de 25 redes neurais MLP jogadoras que, apesar de apresentar arquitetura similar à do IIGA, diferem dele em aspectos importantes do treinamento, conforme discutido no Capítulo 6.

EGA - *End Game Agent*: o EGA é o agente de final de jogo escolhido pela Rede Kohonen-SOM, dentre os 25 agentes de final de jogo existentes, para atuar na partida em estados finais do jogo de Damas. Esse agente, por ser o mais próximo do estado atual do tabuleiro, é o que tem maior condição de conduzir a partida, visto que o mesmo foi treinado em estados de tabuleiro similares ao estado corrente avaliado pelo sistema.

O capítulo 5 a seguir apresenta o módulo Redes Kohonen-SOM. Os demais módulos, detalhando as características e estruturas de cada agente que integra o *D-MA-Draughts*, serão apresentados no Capítulo 6.

Capítulo 5

Rede Kohonen-SOM

Conforme apontado na seção 4.1 a utilização da rede Kohonen-SOM do *D-MA-Draughts* possui duas finalidades: *clusterização* dos tabuleiros de final de jogo acompanhada da geração de um agente de final de jogo para cada *cluster* produzido e indicação do EGA.

A seguir é apresentada a arquitetura geral da rede Kohonen-SOM adotada no presente trabalho.

5.1 Arquitetura da rede Kohonen-SOM

A Figura 5.1 ilustra a arquitetura básica da rede Kohonen-SOM do *D-MA-Draughts*. A entrada da rede recebe um estado de tabuleiro de final de jogo representado como vetor de atributos (representação por *features*) obtidos através do mapeamento NET-FEATUREMAP.

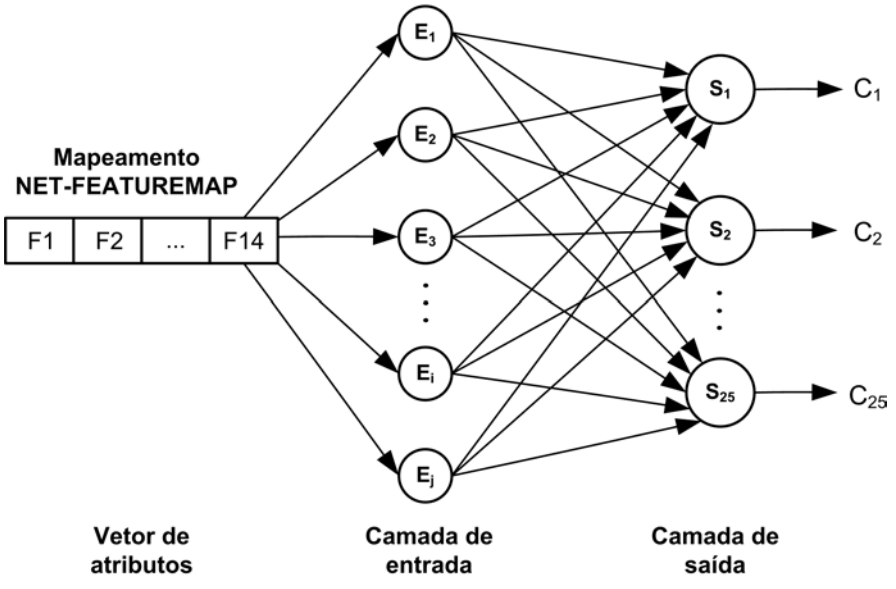

FIGURA 5.1: Arquitetura da Rede Kohonen-SOM do *D-MA-Draughts*

O mapeamento NET-FEATUREMAP converte a representação vetorial do estado do tabuleiro de final de jogo B em uma representação baseada em um vetor de atributos construído a partir das 28

características propostas por Samuel [6] listadas na tabela 2.2. Assim, conforme a seção 2.10.2, o mapeamento C do *D-MA-Draughts* é definido como C: B $\longrightarrow \mathbb{N}^{28}$, onde B é o conjunto de estados de tabuleiro de final de jogo e cada elemento da 28-tupla que corresponde a um tabuleiro B de final de jogo representa a quantidade de peças em B que satisfazem a característica (*feature*) que aquele elemento representa.

A saída da rede é formada por 25 neurônios, onde cada um corresponde a um *cluster* que agrupa estados finais de tabuleiro em função de suas similaridades (a justificativa para o número de *clusters* - no caso, 25 - será apresentada na seção 5.2.1). Todos os nodos da camada de entrada são conectados a todos os neurônios da camada de saída. Quando um vetor é apresentado à entrada da rede Kohonen-SOM é calculada a Distância Euclidiana entre o estado apresentado à entrada da rede e cada neurônio da camada de saída. O neurônio de saída S_i que for mais próximo do tabuleiro de entrada será o vencedor da competição. Tal vitória, em função da finalidade com a qual a Rede Kohonen-SOM está sendo utilizada, definirá um dos seguintes elementos: o *cluster* C_i correspondente ao vencedor S_i ao qual pertencerá o estado de entrada (no caso de a rede Kohonen-SOM estar sendo usada com a primeira finalidade) ou a rede MLP_i (associada a S_i) que será usada como EGA (no caso de a rede Kohonen-SOM estar sendo usada durante um jogo que tenha chagado a um estado final cumprindo sua segunda finalidade).

As próximas seções descrevem o processo de operação da Rede Kohonen-SOM em cada uma de suas finalidades no *D-MA-Draughts*: processo de treinamento como ferramenta de *clusterização* e processo de definição do EGA.

5.2 1ª Finalidade: Treinamento da Kohonen-SOM para Geração de Agentes de Final de Jogo

O módulo *Agentes de Final de Jogo*, presente na arquitetura do jogador *D-MA-Draughts* descrita na seção 4.1, é formado por 25 redes neurais MLP. Cada uma é treinada de modo a representar um dado "perfil" de tabuleiro de final de jogo definido em um processo de *clusterização* por meio de uma rede Kohonen-SOM. Esta seção apresenta o processo realizado a fim de gerar os *clusters* referentes aos agentes de final de jogo do *D-MA-Draughts*: a subseção 5.2.1 descreve a arquitetura do processo de obtenção dos *clusters*; e, a subseção 5.2.2, apresenta a descrição do algoritmo de *clusterização* utilizado.

5.2.1 Arquitetura do Processo de Geração dos Agentes de Final de Jogo

Os *clusters* do *D-MA-Draughts* são compostos por estados de tabuleiro de final de jogo. Tais estados foram obtidos a partir de uma base de dados contendo 4000 estados de tabuleiro de final de jogo. A arquitetura geral do processo de *clusterização* da base de dados do jogador *D-MA-Draugths* é apresentada na Figura 5.2.

Os módulos que compõem essa arquitetura são:

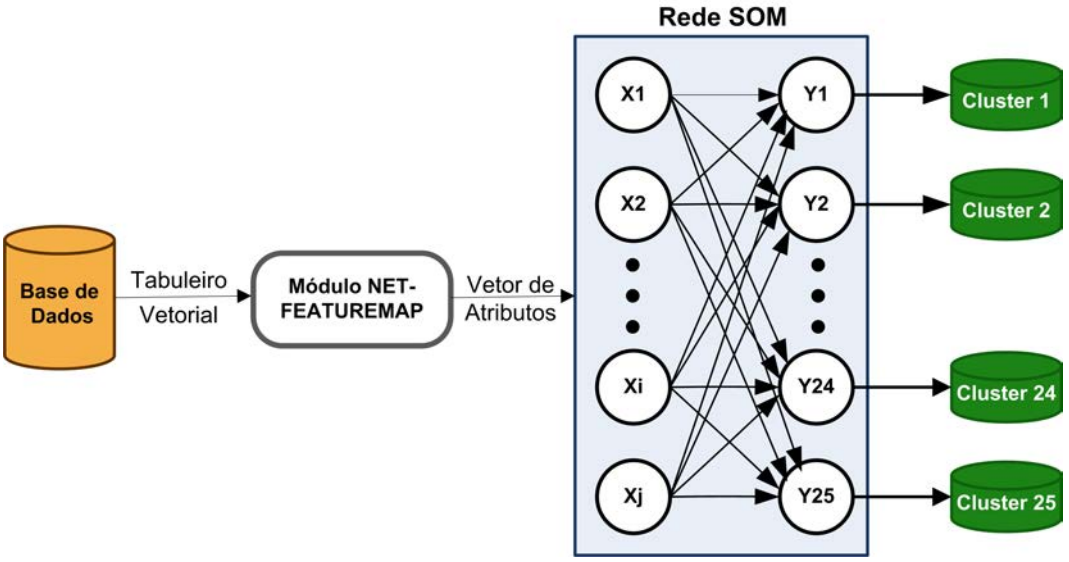

FIGURA 5.2: Arquitetura do processo de *Clusterização*

Base de dados: composta por 4000 estados de tabuleiros de final de jogo (tabuleiros com 12 peças) obtidos de torneios entre os jogadores automáticos de Damas *VisionDraughts*, *NeuroDraughts*, *LS-Draughts* e *IIGA* do *MP-Draughts*. Essa base contém os estados de tabuleiros que serão usados no processo de treinamento dos agentes especialistas em final de jogo. Os tabuleiros presentes nessa base são representados na forma vetorial (descrito na seção 2.10.1), visto que este tipo de representação é simples e eficiente, além disso, os estados de tabuleiro foram obtidos de partidas envolvendo jogadores que utilizam dessa representação durante os jogos;

Módulo NET-FEATUREMAP: este módulo é o responsável por converter um tabuleiro vetorial (recebido do módulo Base de Dados) em um vetor de atributos e enviá-lo ao módulo Rede Kohonen-SOM;

Rede Kohonen-SOM: nesta fase, a rede Kohonen-SOM do *D-MA-Draughts* é treinada para atuar como ferramenta de *clusterização* de tabuleiros de final de jogo e como geradora dos agentes de final de jogo. A saída da rede é composta por 25 neurônios, cada um representando um dos *clusters* de final de jogo produzido por ela. O algoritmo de *clusterização* será apresentado na seção 5.2.2.

Clusters: este módulo é formado por 25 *clusters*, onde cada um é representado por um neurônio de saída da rede Kohonen-SOM. Os estados de um dado *cluster* devem ter alto grau de similaridade entre si, ou seja, possuem várias características de tabuleiro em comum. Adotou-se o total de 25 *clusters*, pois foi observado que quando era tratado um número menor de grupos havia estados muito díspares agrupados em um mesmo *cluster*. Além disso, quando era tratado um número maior de *clusters*, notou-se que havia grupos com um número insignificante de estados de tabuleiro, chegando, em alguns casos, a extremos em que foram identificados *clusters* vazios. Por isso, após observar estes fatores, concluiu-se que 25 *clusters* era um número que minimizava os problemas citados. Ao final da execução do processo de *clusterização*, a cada *cluster* será associada uma rede MLP (agente de final de jogo). O treinamento dessas redes MLP será apresentado no Capítulo 6.

Em resumo, o fluxo do processo de *clusterização* ocorre da seguinte maneira: a base de dados passa como parâmetro de entrada para o módulo NET-FEATUREMAP os estados de tabuleiro de final de jogo na forma de representação vetorial. É passado um estado de tabuleiro de cada vez. O módulo NET-FEATUREMAP converte o tabuleiro de final de jogo, recebido em forma vetorial, para a forma de representação por *features* que é passado como entrada para a rede Kohonen-SOM. É conectada cada posição do vetor a um nodo específico da camada de entrada da rede Kohonen-SOM. Em seguida, é verificado, segundo a regra de similaridade (Distância Euclidiana), qual neurônio de saída mais se assemelha com o vetor de atributos (características/*features*) e elege-o como neurônio vencedor. Tal neurônio representa um dos 25 *clusters*, desta forma, o *cluster* correspondente ao neurônio vencedor receberá o estado de tabuleiro que foi apresentado a entrada da rede.

A próxima seção descreve como foi formada a base de 4000 estados de tabuleiros de final de jogo.

5.2.1.1 Obtenção da Base de Dados a partir da qual são gerados os Agentes de Final de Jogo

Para formação dos *clusters* do *D-MA-Draughts* foi gerada uma base de dados contendo 4000 estados de tabuleiro de final de jogo obtidos em jogos reais disputados por outros jogadores de Damas automáticos. A escolha de usar tabuleiros com 12 peças para representar fases de final de jogo foi feita após a observação de vários estágios de um jogo de Damas. No estágio do jogo com 12 peças sobre o tabuleiro, ainda não existe um jogador tendencioso a ganhar o jogo, ao contrário do que pode ser observado em tabuleiros com 10 ou menos peças. Em tabuleiros com 10 peças ou menos, a vantagem de um jogador em relação ao seu oponente, na maioria dos casos, já é visível e a possibilidade de reversão de resultado de jogo fica comprometida, o que não é interessante para os agentes de final de jogo. Desse modo, o *D-MA-Draughts* utiliza para representar fases de final de jogo, tabuleiros com 12 peças.

Em se tratando de tabuleiros de 12 peças, nota-se que existe cerca de 1 quatrilhão de configurações possíveis de tabuleiros [25], como pode ser observado a Figura 5.3. Assim sendo, mesmo considerando os mais modernos recursos computacionais, é impraticável treinar agentes para serem especialistas em todos esses estados.

Como o número de configurações possíveis de tabuleiros com 12 peças é muito grande, e entre essas configurações existem tabuleiros cuja ocorrência é praticamente impossível de acontecer durante um jogo (tabuleiros com 12 peças rainhas, por exemplo), foram realizados torneios entre alguns jogadores automáticos de Damas na intenção de verificar quais estados de tabuleiro com 12 peças são mais prováveis. Tais torneios envolveram os jogadores *NeuroDraughts* [28], [16], *LS-Draugths* [17], [15], *VisionDraughts* [19], [18] e IIGA do *MP-Draughts* [20], [21]. Observou-se que os estados alcançados por esses jogadores são realistas e tem uma boa chance de ocorrerem durante uma partida de jogo de Damas, isso porque esses estados foram obtidos durante partidas reais de jogo de Damas.

O torneio entre os jogadores foi distribuído da seguinte forma:

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

FIGURA 5.3: Espaço de estados para o jogo de Damas: quantidade de estados possíveis de acordo com o número de peças sobre o tabuleiro [4]

- 1000 jogos entre *VisionDraughts* e *NeuroDraughts*
- 1000 jogos entre *VisionDraughts* e *LS-Draughts*
- 1000 jogos entre *VisionDraughts* e *IIGA*
- 1000 jogos entre *LS-Draugths* e *NeuroDraughts*

Inicialmente foi gerada uma base de dados contendo 1000 estados de tabuleiros de final de jogo obtidos de jogos entre esses jogadores. Porém, essa quantidade não foi suficiente para obter uma boa distribuição dos estados entre os *clusters*, de modo que alguns *clusters* se encontravam vazios ao final do processo de *clusterização*. Foram feitos também testes com bases de 2000 e 3000 estados, porém os melhores resultados foram obtidos com 4000 estados.

Após o término do torneio, a base de dados com 4000 estados de tabuleiros já está montada. No intuito de selecionar quais estados de tabuleiro devem compor um determinado *cluster* realiza-se o processo de *clusterização* descrito na arquitetura apresentada em 5.2.1. A próxima seção apresenta o algoritmo de *clusterização* utilizado pelo *D-MA-Draughts*.

5.2.2 Algoritmo de Clusterização

O *D-MA-Draughts* utiliza o algoritmo Kohonen-SOM para agrupar os estados de tabuleiro de final de jogo presentes na base de dados. A medida de similaridade usada é o quadrado da Distância Euclidiana.

Visando simplificar o entendimento do algoritmo de *clusterização* Kohonen-SOM, é apresentado, no pseudocódigo 1, tal algoritmo juntamente com uma breve descrição de cada linha.

Ressalta-se que entre as linhas 3 e 15 é realizada a etapa de treinamento em que os pesos são reajustados de modo que a rede aprenda a *clusterizar* estados de final de jogo. E, entre as linhas 16 e 22 é constituída a etapa em que a rede Kohonen-SOM, já treinada, classifica os 4000 estados de final de jogo da base de dados original, segundo sua similaridade, em 25 novas bases de dados, cada uma associada a um dos *clusters* de saída. Os tabuleiros de cada uma dessas novas bases serão usados como tabuleiros iniciais de treinamento da MLP associada ao *cluster* a que tal base se refere, conforme detalhado no Capítulo 6.

Pseudocódigo 1 Algoritmo de clusterização por redes Kohonen-SOM

```
1: Kohonen(int:radius, double:alpha, int:numberNeuronsIn, int:numberNeuronsOut)
2: Initialize weight W_ij
3: while alpha > 0.1 do
4:    for each input datum B_k do
5:       for each j, compute do
6:          D(j) = sum_{i=1}^{n} (W_ij - A_i)^2
7:       end for
8:       Find Y_j such that D(J) is a minimum
9:       for all units j within a specified neighborhood of J, and for all i do
10:         W_ij(new) = W_ij(old) + alpha [X_i - W_ij(old)]
11:      end for
12:   end for
13:   Update learning rate
14:   Test stopping condition
15: end while
16: for each input datum B_k do
17:   for each j, compute do
18:      D(j) = sum_i (W_ij - A_i)^2
19:   end for
20:   Find Y_j such that D(J) is a minimum
21:   Store B_k in Y_j
22: end for
```

Linha 1: o algoritmo *Kohonen-SOM* recebe o raio que indica o número de vizinhos do neurônio de saída que está sendo avaliado cujos os vetores de pesos dos vizinhos serão reajustados juntamente com o neurônio de saída; a taxa de aprendizagem do algoritmo, o número de nodos da camada de entrada da rede (igual ao número de características implementadas) e o número de neurônios da camada de saída (igual ao número de *clusters* que devem ser gerados no final da execução do algoritmo). No caso específico deste algoritmo, o raio da vizinhança é 1, a taxa de aprendizagem começa em 1 e sendo decrementada em 5% ao longo do treinamento, o número de neurônios de saída é 25 (número de *clusters* gerado no final do algoritmo). Ao final da execução do algoritmo, os vetores associados a cada neurônio devem ser armazenados em 25 arquivos diferentes, cada um representando um neurônio de saída específico;

Linha 2: cada conexão W_{ij} no mapa de Kohonen, mostrado na Figura 2.12 no capítulo 2, tem um peso associado. A rede é inicializada com valores aleatórios para esses pesos. Os valores dos pesos são atualizados durante o treinamento da rede. Cada nó vencedor atualiza seu próprio peso e, em menores graus, os nós da vizinhança também são atualizados;

Linha 3: a taxa de aprendizagem *alpha* é a condição de parada do algoritmo. Os pesos da rede são atualizados até que a taxa de aprendizagem seja menor que 0.1. Quando a taxa é atingida, considera-se que o algoritmo já convergiu e a rede está apta a classificar corretamente a base de dados;

Linha 4: o laço de execução é repetido até que todos os estados de tabuleiro B_k presentes na base de dados tenham sido analisados pela rede Kohonen-SOM. Para cada estado B_k, execute os passos de 5 a 12 do algoritmo;

Linha 5: dado um estado de tabuleiro, representado por características (C: B $\rightarrow \mathbb{N}^{28}$ B_k, o quadrado da distância Euclidiana D_j entre esse tabuleiro e cada neurônio Y_j da camada de saída deve ser analisado, sendo que j corresponde ao neurônio de saída que está sendo analisado;

Linha 6: o cálculo do quadrado da Distância Euclidiana $D_{(j)}$ entre o vetor de pesos que representa o neurônio de saída Y_j e o vetor de entrada da rede neural é calculado com base na fórmula: $D_{(j)} = \sum_{i=1}^{n} (W_{ij} - A_i)^2$, onde n representa qual nó de entrada X_i está sendo analisado, o índice j especifica o neurônio de saída, W_{ij} é o valor do peso correspondente à conexão entre i e j, e A_j corresponde ao valor do atributo representado no nodo X_i da camada de entrada da rede neural. Como explicado anteriormente, cada posição do vetor característica é conectada a um nodo específico da camada de entrada, ou seja, um vetor de características com 28 posições é representado da seguinte maneira na entrada da rede Kohonen-SOM: B_k = $<A_1, A_2, ...,A_{27}, A_{28}>$. Repete-se esse processo até que todas as distâncias entre um dado vetor de entrada e todos os neurônios de saída sejam calculados;

Linha 8: após calculadas todas as distâncias entre o vetor de entrada e os neurônios de saída Y_j, encontra-se o neurônio de saída Y_j com menor distância;

Linhas 9 e 10: o neurônio Y_j com menor distância tem seus pesos atualizados, assim como os seus vizinhos, sendo atualizado um vizinho à direita e um à esquerda deste neurônio. A atualização dos pesos é feita usando a seguinte fórmula: $W_{ij}(new) = W_{ij}(old) + \alpha[X_i - W_{ij}(old)]$, em que $W_{ij}(old)$ é o peso antigo do neurônio. Após o ajuste de pesos, esses neurônios estarão mais parecidos com a entrada oferecida;

Linha 13: terminada a classificação de todos os estados de tabuleiros B_k presentes na base, a taxa de aprendizagem deve ser ajustada com base no decremento estabelecido no começo do algoritmo (5%), ou seja, $alpha = alpha * 0.05$;

Linha 14: ajustada a taxa de aprendizagem, testa-se a condição de parada do algoritmo, de modo que, se ela não tiver sido atingida, os passos de 4 a 12 serão executados novamente. Caso a taxa de aprendizagem já tenha sido atingida, a rede estará apta a classificar corretamente os estados da base. A partir desse momento, não serão mais efetuadas quaisquer alterações

nos pesos da rede e ela passa da fase de treinamento para a fase de utilização efetiva. O processo de classificação e geração dos arquivos com os tabuleiros classificados acontece da linha 16 a 22 do algoritmo;

Linha 16: para cada estado de tabuleiro B_k presente na base de dados, as linhas 17 a 19 são executadas na intenção de se descobrir qual neurônio de saída melhor representa o estado;

Linhas de 17 a 19: dado um estado B_k, é calculada a distância entre tal estado e cada neurônio da rede no intuito de descobrir a qual neurônio de saída o estado B_k deve pertencer. A fórmula usada para verificar a qual neurônio o estado deve ser relacionado é similar à apresentada na linha 6 deste algoritmo. A diferença é que, depois de encontrado o neurônio que representará o estado, tanto o peso de tal neurônio como os pesos de seus vizinhos não são reajustados. Os pesos dos neurônios de saída devem ser reajustados somente durante o processo de treinamento da rede;

Linha 20: dado que já tenham sido calculadas as distâncias entre o tabuleiro de entrada B_k e todos os neurônios de saída, o neurônio de saída Y_j "vencedor" que representará o estado de tabuleiro será o que possuir o menor valor do quadrado da Distância Euclidiana D_j;

Linha 21: nesta linha do algoritmo, o estado de tabuleiro B_k é armazenado no *cluster* Y_j. Quando o estado B_k é direcionado para o *cluster*, sua representação por característica é abandonada, sendo que o estado armazenado no *cluster* é o estado de representação vetorial (veja seção 2.10.1). O estado de tabuleiro representado por característica é usado somente para definir em qual *cluster* um dado estado deve ser armazenado.

5.3 2ª Finalidade: Processo de seleção do EGA pela Kohonen-SOM

O processo de seleção do EGA em fases de final de jogo é ilustrada na Figura 5.4.

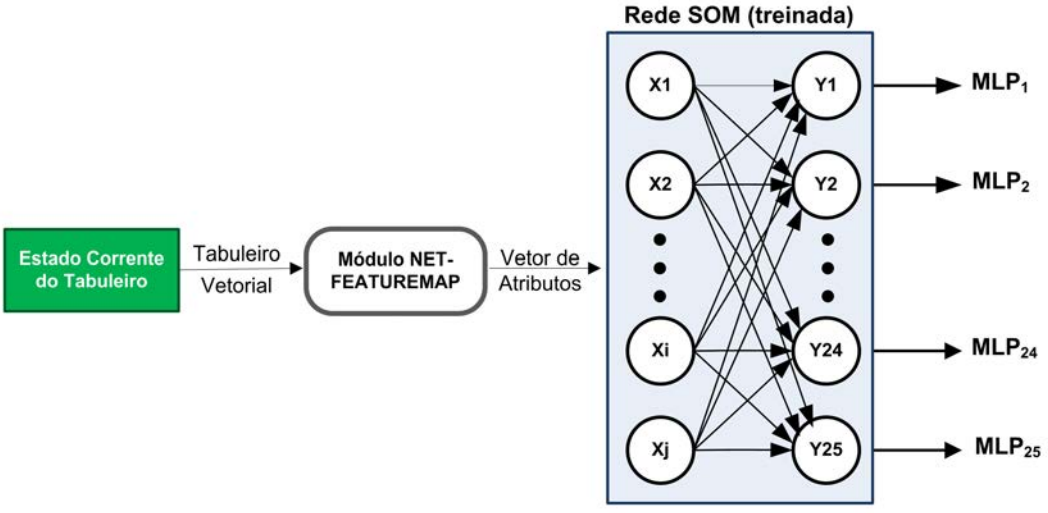

FIGURA 5.4: Arquitetura da Rede Kohonen-SOM no processo de seleção do EGA

Os módulos que compõem essa arquitetura são semelhantes ao do processo de treinamento da rede Kohonen-SOM apresentado na seção 5.1 salvo que agora não há atualização de pesos, uma vez que a Rede Kohonen-SOM já está treinada. O módulo de entrada desta arquitetura também sofre alterações, visto que não há mais uma base de dados de tabuleiro de final de jogo para ser tratada e, sim, apenas um tabuleiro de final de jogo que será avaliado a fim de verificar qual EGA mais se identifica com o referido estado. Assim sendo, tal tabuleiro, que faz uso de representação vetorial, é apresentado ao módulo NET-FEATUREMAP para conversão em um vetor de atributos (vetor de características). Este vetor será apresentado à entrada da rede Kohonen-SOM. A saída da rede será direcionada ao neurônio que possuir menor Distância Euclidiana comparado aos demais. Desta forma, será possível identificar a qual *cluster* o estado corrente de final de jogo mais se assemelha. Sabendo qual é o identificador deste *cluster*, será possível verificar qual MLP (EGA) foi treinada para lidar com este tipo de estado de tabuleiro, assim sendo, o EGA selecionado dará prosseguimento à partida.

A próxima seção apresenta o algoritmo utilizado no processo de definição do EGA.

5.3.1 Algoritmo de Definição do EGA

O pseudocódigo 2 apresenta o processo de definição do EGA seguido de uma breve descrição de cada linha [20].

Pseudocódigo 2 Rotina que trata a seleção do agente de final de jogo (EGA)

```
1: SelectEGA(board:boardState)
2: for each EndGameNetwork in set of EngGameNetworks do
3:     d[i] = distancia(boardState, EndGameNetworks[i])
4: end for
5: j = i of minimum(d[i])
6: return j
```

Linha 1 : o algoritmo Select-EGA recebe, como argumento, o estado corrente do tabuleiro de final de jogo, *boardState*;

Linha 2 : para cada um dos agentes especializados em final de jogo que compõe o *D-MA-Draughts*, executa-se a linha 3 do algoritmo, de modo a encontrar o agente que substituirá o IIGA na partida;

Linha 3 : calcula-se o quadrado da Distância Euclidiana entre o estado corrente do jogo(tabuleiro de final de jogo) e cada um dos 25 agentes de final de jogo. Tal distância é calculada entre as características presentes no tabuleiro corrente e as características presentes no agente em questão. O valor resultante desse cálculo é armazenado na variável *d[i]*;

Linha 5 : depois de calculado o quadrado da Distância Euclidiana entre o estado corrente e os agentes de final de jogo, verifica-se qual das distâncias é a menor. O valor da variável i corresponde à posição onde se encontra a menor distância e indica o agente de final de jogo EGA que deverá ser selecionado;

Linha 6 : depois de definido o EGA, o valor referente a ele é retornado para a função principal. A função principal transfere o jogo para o referido agente.

O presente trabalho analisa o desempenho da plataforma distribuída multiagente do *D-MA-Draughts* em duas dinâmicas distintas de jogo que diferem entre si pela maneira como são utilizados os agentes de final de jogo, conforme explicado na próxima seção.

5.3.2 Dinâmicas de jogo do D-MA-Draughts: do IIGA para o EGA

O desempenho do *D-MA-Draughts* é avaliado a partir de duas dinâmicas de atuação no jogo. Tais dinâmicas ocorrem através da comunicação entre os agentes do sistema. Basicamente, esta comunicação consiste na transferência do estado corrente do tabuleiro (após o agente atuante na partida executar uma ação) para o agente que assumirá o jogo. Desta forma, os agentes do *D-MA-Draughts* coordenam suas ações de modo cooperativo em prol de um objetivo, conforme abordado na seção 2.1.1. As subseções a seguir descrevem cada uma dessas dinâmicas de atuação nos jogos. Os resultados desta análise serão abordados no Capítulo 7.

5.3.2.1 Dinâmica de Jogo I

A primeira dinâmica de jogo adotada pelo *D-MA-Draughts* é ilustrado na Figura 5.5 (note que esta dinâmica também foi usada no *MP-Draughts* que antecedeu a presente proposta). O agente IIGA inicia a partida e a conduz até que o tabuleiro tenha, no mínimo, 13 peças. Quando o tabuleiro atinge no máximo 12 peças é caracterizado um estado de tabuleiro de final de jogo. A partir deste momento, este estado será enviado para a rede Kohonen-SOM que avaliará qual agente de final de jogo é o mais apto para se tornar o EGA da partida. Desta forma, uma vez definido o EGA este conduzirá a partida até o final.

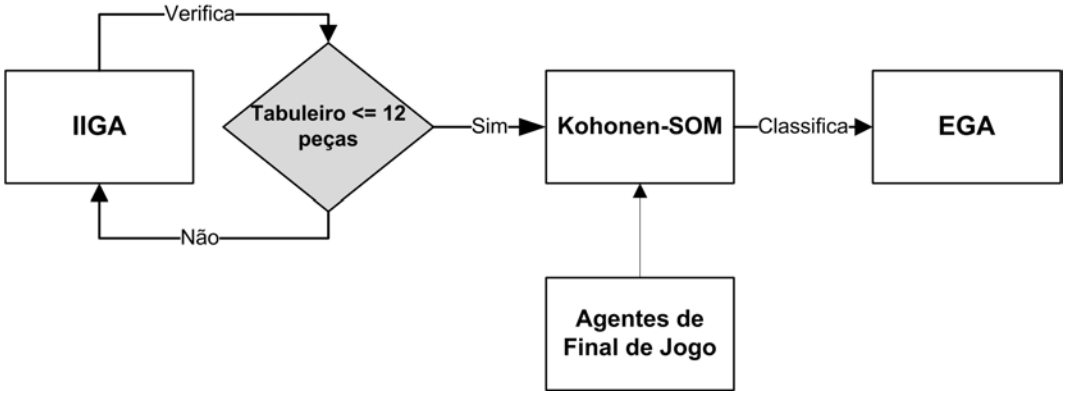

FIGURA 5.5: Dinâmica de jogo I do *D-MA-Draughts*

5.3.3 Dinâmica de Jogo II

A segunda dinâmica de jogo adotada no *D-MA-Draughts* é ilustrada na Figura 5.6. Como é possível observar, a dinâmica de jogo II consiste de uma pequena alteração na dinâmica de jogo I

(seção 5.3.2.1), ou seja, o fluxo da partida é o mesmo até que seja atingido um estado de tabuleiro de final de jogo e um EGA seja definido. Desta forma, haverá uma interação na escolha do EGA, de modo que, a cada nova jogada a ser executada pelo *D-MA-Draughts*, a rede Kohonen-SOM verificará dentre os agentes de final de jogo o mais apto a se tornar o EGA e prosseguir a partida. Esta alteração foi realizada a fim de avaliar a atuação de todos os agentes de final de jogo, visto que, após alguns movimentos, outro EGA poderá se identificar melhor com o estado corrente do tabuleiro.

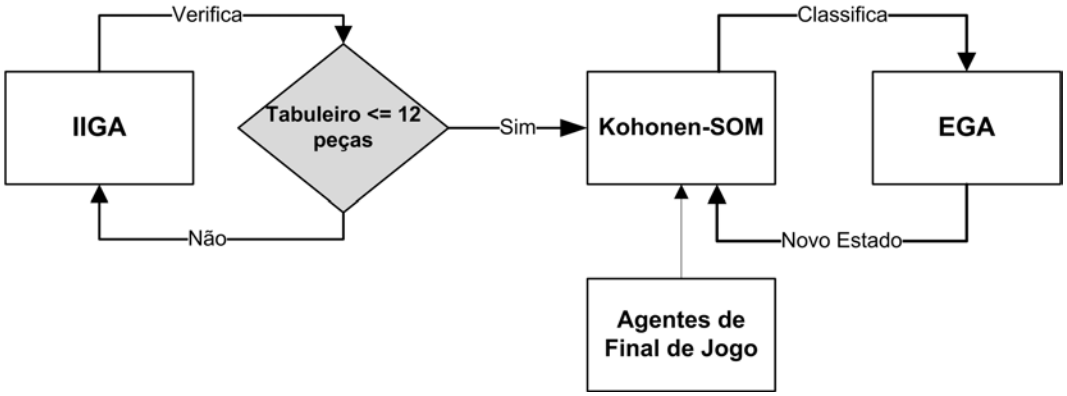

FIGURA 5.6: Dinâmica de jogo II do *D-MA-Draughts*

Capítulo 6

Técnicas e Estruturas dos Agentes do DMA-Draughts

Os agentes que compõem o *D-MA-Draughts* se baseiam na arquitetura do jogador automático *D-VisionDraughts* [22], ou seja, cada agente é implementado como uma rede neural multicamadas (*Multi-Layer Perceptron* - MLP) que opera em um ambiente de alta performance, que, distintamente do atual campeão mundial homem-máquina em Damas, Chinook [4], aprende sem supervisão humana. Os pesos da rede são atualizados pelos métodos das Diferenças Temporais TD(λ) utilizando a técnica *self-play* com clonagem. A escolha do melhor movimento é conduzida pelo algoritmo de busca paralelo YBWC. Para isso, cada folha da árvore de busca do jogo, obtida a partir da expansão do estado corrente do tabuleiro, tem sua representação convertida para o mapeamento NET-FEATUREMAP (seção 2.10.2) e é apresentada na entrada da rede neural para ser avaliada. A próxima seção traz em detalhes a arquitetura dos agentes do *D-MA-Draughts*.

6.1 Arquitetura individual e ciclo de operação dos agentes do D-MA-Draughts

A Figura 6.1 ilustra a arquitetura geral e o ciclo de operações executadas por cada agente do *D-MA-Draughts* a cada vez que ele efetua uma tomada de decisão para a escolha de um movimento a partir do estado corrente do jogo. Ressalta-se que o ciclo apresentado sofre pequenas alterações em função de ser executado em jogos-treino ou em jogos normais de disputa, conforme explicado a seguir. Ambos os tipos de jogo executam os passos de #1 a #6, os demais passos (de #7 a #13) somente são executados em jogos-treino. Particularmente, a dinâmica dos jogos-treino será detalhada na seção 6.3.3.

Os passos da Figura 6.1 são:

Passo #1 : O estado corrente do tabuleiro (chamado aqui T_x) é apresentado ao *Módulo de Busca*. Assim sendo, T_x foi obtido a partir da execução do último movimento m_x. Além disso, P_x representa a avaliação produzida pela MLP para T_x no ciclo anterior do treinamento.

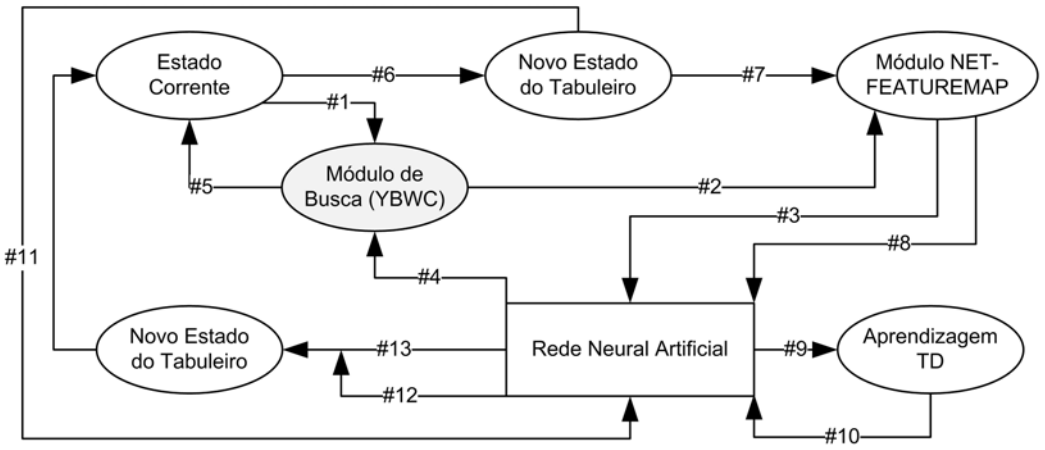

FIGURA 6.1: Arquitetura geral dos agentes do D-MA-Draughts

Passo #2 : Este módulo realiza uma busca em profundidade limitada ou uma busca com profundidade iterativa, utilizando o algoritmo de busca paralelo YBWC, a partir do estado de tabuleiro T_x. As folhas da árvore de jogo são repassadas ao mapeamento NET-FEATUREMAP.

Passo #3 : Cada folha da árvore de busca do jogo corresponde a um estado de tabuleiro que, representado por NET-FEATUREMAP, é apresentado à camada de entrada da rede MLP para ser avaliado.

Passo #4 : Os valores calculados no Passo #3 são retornados ao *Módulo de Busca* para cálculo do melhor movimento m_y a ser executado.

Passo #5 : O movimento m_y é calculado e retornado ao sistema para que ele o execute a partir do estado corrente T_x.

Passo #6 : O movimento m_y sugerido pelo Passo #5 é executado e o estado corrente do tabuleiro é modificado para T_y (novo estado do tabuleiro). Em jogos normais de disputa, o ciclo de escolha do melhor movimento termina neste passo (a partir daí, cabe ao oponente escolher o movimento no estado T_y). Em jogos-treino, o ciclo prossegue nos passos a seguir.

Passo #7 : O estado de tabuleiro T_y é convertido através do *Módulo NET-FEATUREMAP*.

Passo #8 : O estado T_y é apresentado à camada de entrada da rede MLP para ser avaliado produzindo como resultado P_y.

Passos #9 e #10 : A avaliação P_y que foi calculada pela rede MLP para T_y e a avaliação P_x de T_x são usadas pelo *Módulo de Aprendizagem TD* como parâmetros de entrada para calcular os novos pesos da rede MLP. São usados também como parâmetros as avaliações dos estados gerados pela execução dos movimentos anteriores a m_x, desde o início do jogo, o que caracteriza a propriedade dos Métodos das Diferenças Temporais de considerar o impacto de cada ação m_i executada ao longo do jogo.

Passo #11 : O estado de tabuleiro T_y é avaliado novamente, todavia, utilizando os pesos atualizados da rede MLP.

Passo #12 : O valor da avaliação obtido no Passo #11 produz um novo valor de avaliação $P_y^{'}$ para o estado de tabuleiro T_y.

Passo #13 : O valor $P_y^{'}$ produzido no Passo #12 corresponderá à avaliação do agora estado corrente de jogo T_y a ser usado no cálculo de reajuste de pesos que será efetuado no próximo ciclo em que o agente deverá, novamente, escolher um movimento (após o oponente ter executado um movimento em T_y).

Desta forma, os módulos que constituem a arquitetura do *D-MA-Draughts* podem ser resumidos em:

Módulo de Busca: este módulo é responsável pela escolha do melhor movimento a ser executado pelo agente, em função do estado corrente do tabuleiro, através do algoritmo de busca paralelo YBWC. Tal estado é recebido como parâmetro pelo referido módulo na forma vetorial.

Rede Neural Multicamadas: a rede neural recebe, em sua camada de entrada, um estado de 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;

Módulo NET-FEATUREMAP: por meio do mapeamento das características utilizadas pelo sistema, esse módulo converte a representação vetorial do estado corrente que lhe é repassado pelo módulo de busca em representação NET-FEATUREMAP, apresentando esta última à entrada da rede neural MLP para avaliação. O *D-MA-Draughts* utiliza até 16 características para representação dos estados do tabuleiro em seu mapeamento NET-FEATUREMAP (conforme descrito na seção 6.3). Desta forma, o mapeamento C é C:B $\rightarrow \mathbb{N}^{16}$. Cada característica tem um valor absoluto que representa a sua medida analítica em 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 representada na entrada da rede neural.

Módulo de Aprendizagem: o ajuste de pesos da rede neural é feito pelo método das Diferenças Temporais TD(λ). Esse processo é responsável pela aquisição de conhecimento do sistema.

Ressalta-se que a fase de treinamento do IIGA e dos agentes de final de jogo diferem em alguns aspectos como será mostrado na seção 6.3.4, contudo, todos eles apresentam a mesma arquitetura e os mesmos ciclos de treinamento apresentados acima.

A seção 6.2 apresenta o Módulo de Busca do *D-MA-Draughts*. A seção 6.3 apresenta o Módulo de Aprendizagem, o qual envolve a descrição do Módulo NET-FEATUREMAP e do Módulo MLP (Rede Neural Multicamadas).

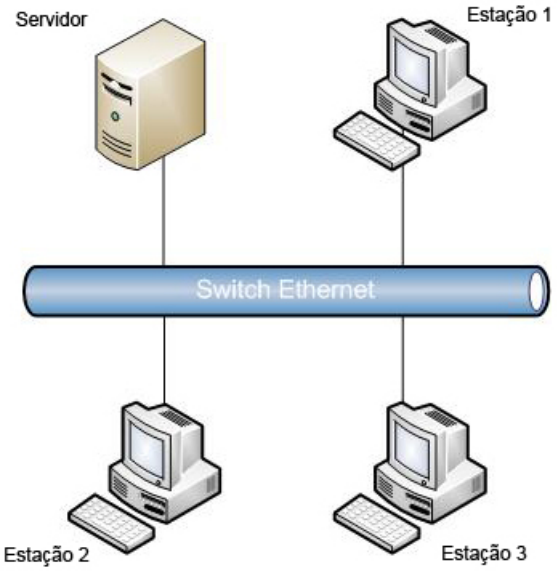

FIGURA 6.2: Infra-estrutura de desenvolvimento dos agentes do *D-MA-Draughts*

6.2 Módulo de Busca

O módulo de Busca do *D-MA-Draughts* é composto pelo algoritmo YBWC (distribuição do algoritmo de busca Alfa-Beta). O YBWC foi escolhido, pois, além de ter obtido grande êxito na distribuição do algoritmo Alfa-Beta com bons resultados na prática [66], foi a opção que melhor atendeu as necessidades do projeto, por ter um conceito simples (diferente do APHID - veja a seção 2.8.4) e ser adequado à arquitetura disponível para os experimentos (máquinas com memória distribuída, o que elimina a possibilidade de utilização do DTS - veja a seção 2.8.3).

No presente trabalho, o desempenho desse algoritmo foi avaliado em situações onde ele conta com um número variável m de processadores, onde $10 \leq m \leq 16$. O objetivo deste aumento de processadores com relação à versão antecessora *D-VisionDraughts* (que conforme a seção 3.2.4, opera com um número fixo de 10 processadores) é proporcionar ao *D-MA-Draughts* um ambiente mais favorável que lhe permita melhorar seu *look-ahead*. Além disso, ainda permite a inclusão de outras melhorias como poderá ser visto na seção 6.3, que detalha o Módulo de Aprendizagem deste sistema, onde o mapeamento NET-FEATUREMAP do *D-MA-Draughts* passa a contar com um número maior de características para representação dos estados do tabuleiro do jogo. A Figura 6.2 ilustra o ambiente de infra-estrutura utilizado pelo *D-MA-Draughts*. Note que há um servidor e três estações. Conforme será visto na seção 6.2.1, o servidor é responsável por operar em todos os módulos do sistema, ao passo que as estações operam apenas no módulo de busca.

A implementação do algoritmo YBWC no Módulo de Busca do *D-MA-Draughts* pode trabalhar tanto em profundidade fixa quanto em profundidade iterativa. Em ambos os casos foi adotado o uso de uma TT. A seção 6.2.1 descreve como o algoritmo YBWC foi implementado no *D-MA-Draughts*; a seção 6.2.3 explica como a TT foi introduzida no sistema e a seção 6.2.4 apresenta o emprego da técnica de aprofundamento iterativo.

6.2.1 YBWC no D-MA-Draughts

O módulo de busca do *D-MA-Draughts* conta com um número variável m de processadores e se baseia na distribuição da árvore de busca através do algoritmo YBWC, ou seja, na avaliação paralela das subárvores (subproblemas) oriundos dessa distribuição pelos m processadores disponíveis.

O YBWC do *D-MA-Draughts* foi baseado na distribuição da versão *fail-soft* do algoritmo de busca Alfa-Beta, visto que, conforme explicado na seção 2.6.2, esta versão permite a integração da busca Alfa-Beta com TT sem que se alterem os resultados que seriam produzidos pelo Minimax nas mesmas situações.

Em relação aos m processadores, cada um é identificado por um número inteiro i denotado como P_i (com $0 \leq i \leq m$). Apenas um processador irá atuar em todos os módulos do jogador e é denominado *Processador Principal*, ou P_0. Os demais processadores P_i ($0 < i \leq m$) são chamados *Processadores Auxiliares* e irão atuar apenas no módulo de busca. Assim, enquanto P_0 processa outras rotinas que não pertençam ao módulo de busca, os *Processadores Auxiliares* permanecem em estado *inativo*. Tão logo P_0 inicie o módulo de busca, ele envia uma mensagem, *InitSearch*, para os *Processadores Auxiliares*, o que os habilita a alterarem seu estado de *inativo* para *ocioso*. Quando os processadores estão ociosos eles requisitam tarefas uns aos outros por meio da mensagem *RequestTask*. Um processador *ocioso* P_j recebe uma tarefa de outro processador P_k, muda seu estado para *trabalhando* e torna-se escravo de P_k. Neste caso, P_k é chamado de mestre de P_j. Quando um escravo conclui sua tarefa, ele envia o resultado para seu mestre por meio da mensagem *ResultMessage* e torna-se *ocioso* novamente. Quando o processo de busca é concluído, P_0 envia uma mensagem de finalização, *EndSearch*, para todos os *Processadores Auxiliares*, que tornam-se, assim, *inativos*. A partir deste momento, o *Processador Principal* irá atuar nos demais módulos do sistema. Desta forma, é possível concluir que P_0, durante um jogo, nunca estará no estado *inativo*. A Figura 6.3 (a) e a Figura 6.3 (b) ilustram os possíveis estados de P_0 e dos *Processadores Auxiliares*, respectivamente.

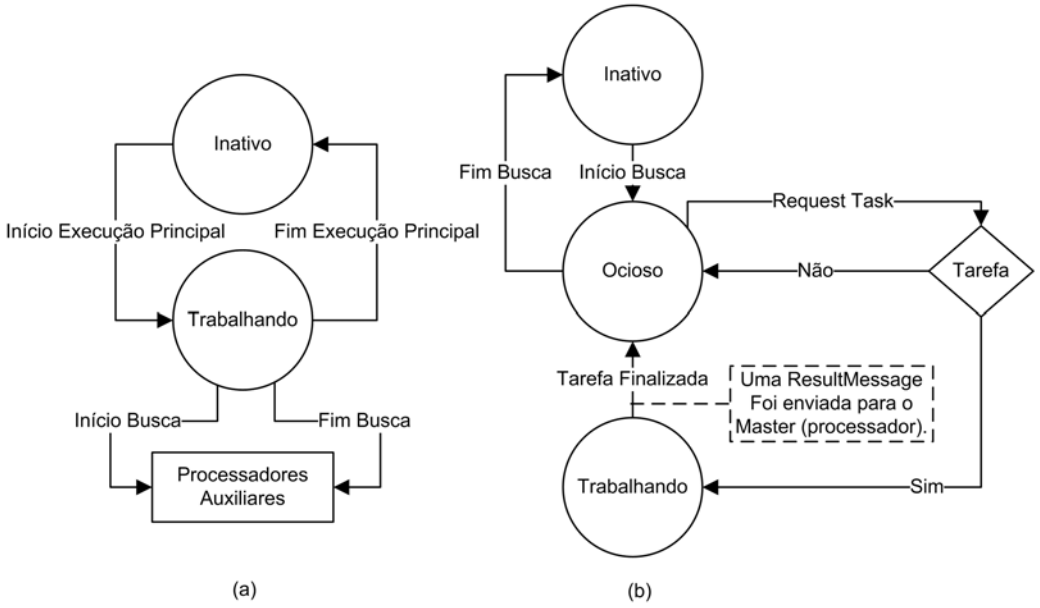

FIGURA 6.3: (a) Estados do Processador Principal; (b) Estados dos Processadores Auxiliares.

O sistema foi testado em duas situações distintas em função da estratégia de exploração da árvore de jogo: profundidade limitada de busca ou aprofundamento iterativo. Para representação do espaço de estados, o *D-MA-Draughts* faz uso de pilhas como estrutura de dados, visto que é possível manter o controle dos nós explorados nestes tipos de busca. Cada processador mantem uma pilha local. A profundidade p da pilha coincide com a profundidade do nó corrente explorado na árvore de busca. Assim, todos os nós explorados em um certo nível de cada subárvore são colocados no mesmo nível da pilha do processador responsável por aquela subárvore. As informações sobre os nós expandidos que são armazenados na pilha correspondem ao caminho na árvore de busca do estado inicial ao estado corrente. Tal caminho é definido como *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 distribuídos, pode haver nós à esquerda ainda em avaliação e nós a direita já avaliados. A Figura 6.4 ilustra este conceito com um exemplo de uma árvore sendo explorada, onde os nós da variação corrente estão em destaque. Note a correspondência entre cada posição p da pilha de estados com a profundidade p da árvore de busca (conforme figura 6.4(b)). Cada posição da pilha tem uma estrutura definida nos moldes do pseudocódigo 3:

Onde, no pseudocódigo 3:

- *alpha e beta*: valores da janela de busca com os quais se inicia a busca em cada nó da variação corrente;
- *low*: limite inferior da busca de cada nó da variação corrente, ou seja, o maior valor dos seus sucessores avaliados, em um nível de maximização;
- *high*: limite superior da busca de cada nó da variação corrente, ou seja, o menor valor dos seus sucessores avaliados, em um nível de minimização;

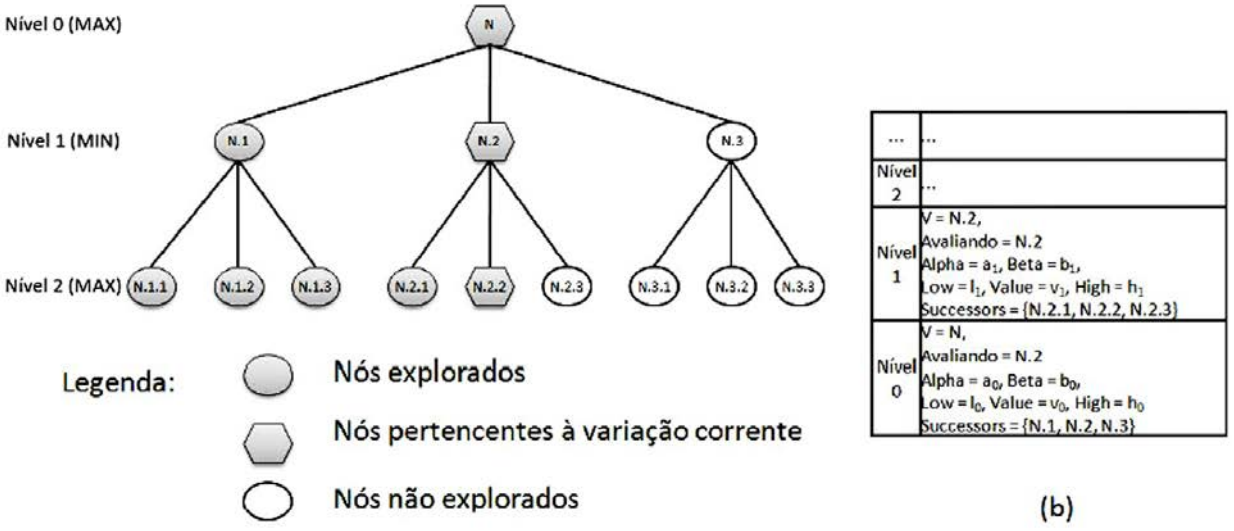

FIGURA 6.4: (a) Árvore de exemplo (b) Ilustração da pilha de expansão do nó N12

Pseudocódigo 3 Estrutra StackEntry

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

- *value*: valor associado a cada nó da variação corrente;
- v: nó (ou estado) da variação corrente;
- *slaves*: lista de escravos. Sempre que um sucessor de v é enviado para avaliação remota em um processador P_k, este processador é adicionado a lista de escravos de v;
- *successors*: 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 4.

Pseudocódigo 4 Estrutura Successors

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

Onde, no pseudocódigo 4:

- *move*: movimento que se deve aplicar ao nó v para obter este sucessor;
- *value*: valor do nó sucessor de v após ser avaliado;
- *status*: o status do sucessor de v. Pode assumir os valores listados no pseudocódigo 5.

Pseudocódigo 5 Status de um nó

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

Onde no pseudocódigo 5:

- *AVAIABLE*: 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 possui status *SOLVED* em um de 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 possuem estado *SOLVED*; ou quando houve uma poda durante sua expansão. Um nó que possui estado *SOLVED* é denominado *completo*.

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

- P_j: número atribuído ao processador sorteado para transferir tarefa a um processador ocioso, P_k, que lho solicita;
- *NP:* número total de processadores disponíveis;
- V_r, ..., 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 Processador Principal podemos ter $r = 0$ (processando a raiz da árvore) e o valor de t nunca deve ultrapassar a profundidade máxima;
- $Alpha_i$, $Beta_i$: 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, marcada pelo envio da mensagem *InitSearch* aos *Processadores Auxiliares*, o estado de tabuleiro a ser avaliado é colocado na pilha de estados do *Processador Principal*. Os *Processadores Auxiliares* que, conforme a Figura 6.3, tornaram-se ociosos, escolhem o processador alvo da mensagem *RequestTask* (denominado P_j) de maneira aleatória. Observe a linha 2 do pseudocódigo 6, onde *NP* é o número de processadores disponíveis. Note que é realizado um sorteio do número do processador P_j ao qual será feita a solicitação. No sorteio, o próprio processador requisitante P_k é excluído da lista de candidatos.

Pseudocódigo 6 Solicitação de Tarefa por um processador ocioso.

```
1: def request_task()
2:     P_j = rand({0, ..., NP} - {P_k})
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. Define-se um nó disponível como um nó v_i da variação corrente, onde i representa a profundidade do nó na árvore de busca de P_j. Um nó $v_i.x \in Sucessores(v_i)$ é dito disponível se as seguintes restrições forem atendidas:

- $x > 1$, pois, conforme explicado na seção 2.8, na avaliação de v_i, a distribuição de tarefas somente ocorre após a avaliação do ramo mais a esquerda de v_i, ou seja, quando $v_i.1$ tem status SOLVED;
- $v_i.x$ não foi e nem está sendo avaliado por nenhum processador.

O processo que trata as mensagens de requisição de tarefas recebidas pelo processador P_j é descrito pelo pseudocódigo 7. Note que P_j percorre os nós de sua variação corrente, partindo da maior profundidade para a menor (linha 5) e, para cada um desses nós, é verificado se há algum nó sucessor disponível (P_j escolherá o sucessor mais profundo e mais à esquerda que não tenha sido avaliado). Se um nó disponível $v_i.x$ 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 a profundidade de v_i na árvore de busca de P_j (linha 13). Se não há nós disponíveis no momento da requisição, a mensagem enviada como resposta, denominada *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 *vi.x*

Pseudocódigo 7 Rotina que trata mensagens de solicitação de tarefa recebidas

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

a ser transferido, tal problema é enviado, a partir da pilha de P_j, juntamente com as seguintes informações adicionais:

- $alpha_i$: Limite inferior inicial da janela de busca em que é disparada a avaliação do nó $vi.x$;
- $beta_i$: Limite superior inicial da janela de busca em que é disparada a avaliação do nó $vi.x$;
- low_i: variável que guarda as atualizações do valor do nível inferior da janela de busca do nó v_i da variação corrente, no caso em que v_i é um nó maximizador;
- $high_i$: variável que guarda as atualizações do valor do nível superior da janela de busca do nó v_i da variação corrente, no caso em que v_i é minimizador;
- i: variável que se refere ao nível em que o nó $v_i.x$ se encontra no processador requisitado P_j.

A relação mestre-escravo entre P_j e P_k 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.

O tratamento aplicado por um mestre P_j ao resultado da avaliação de um nó $vi.x$ processado por um escravo P_k é mostrado no pseudocódigo 8, onde:

Linha 1 : O algoritmo recebe como parâmetros: o nó atual v_i da variação corrente de P_j; a predição $valor_i$ correspondente ao nó v_i; a predição $valor_i.x$ referente a seu filho $v_i.x$ avaliado e retornado por P_k, e o número do processador slave k que está retornando o resultado.

Linha 2 : Esta linha chama a função de remoção do k-ésimo elemento da lista de escravos do processador mestre. Seguindo o exemplo onde P_k se torna escravo de P_j, tão logo P_k lhe retorne o resultado, será removido da lista de escravos da profundidade i da pilha de estados de P_j.

Pseudocódigo 8 Rotina que trata mensagens ResponseResult provenientes do filho P_k recebidas pelo pai P_j

```
1: def response_result_handler(BOARD:v_i, double:value_i, double:value_i.x, int:k):
2: slaves_i.remove(k)
3: if v_i is max-node then
4:    cutoff = false
5:    improvement = false
6:    if value_i.x ≥ beta_i then
7:       cutoff = true
8:    end if
9:    if value_i.x > low_i then
10:      improvement = true
11:   end if
12:   low_i = max(low_i, value_i.x)
13:   value_i = max(value_i, value_i.x)
14:   if cutoff then
15:      send_cutoff(i)
16:   else
17:      if improvement then
18:         send_newbound(i)
19:         update_window_max(i+1,low_i)
20:      end if
21:   end if
22: end if
23: if v_i is min-node then
24:   cutoff = false
25:   improvement = false
26:   if value_i.x ≤ alpha_i then
27:      cutoff = true
28:   end if
29:   if value_i.x < high_i then
30:      improvement = true
31:   end if
32:   high_i = min(high_i, value_i.x)
33:   value_i = min(value_i, value_i.x)
34:   if cutoff then
35:      send_cutoff(i)
36:   else
37:      if improvement then
38:         send_newbound(i)
39:         update_window_min(i+1,high_i)
40:      end if
41:   end if
42: end if
```

Linha 3 : É verificado se v_i é um nó maximizador. Caso afirmativo, executam-se os procedimentos descritos da linha 4 a 21.

Linha 4 : Cria-se a variável *cutoff* do tipo booleano a fim de verificar a eventual ocorrência de uma poda.

Linha 5 : Cria-se a variável *improvement* do tipo booleano a fim de verificar um possível estreitamento da janela alfa-beta.

Linhas 6 e 7 : É verificado se o valor $valor_i.x$ da avaliação do nó $v_i.x$ retornado por P_k é maior ou igual a $beta_i$ (janela superior repassada inicialmente ao nó enviado a P_k). Caso positivo é identificada uma poda beta e o valor da variável *cutoff* é modificado para 'true'.

Linhas 9 e 10 : É verificado se o valor $valor_i.x$ do nó avaliado por P_k é maior que o valor do nível inferior da janela de busca, low_i, do nó v_i da variação corrente. Caso positivo é identificado um estreitamento no limite inferior da janela alfa-beta de v_i, assinalado atribuindo-se o valor 'true' à variável *improvement*;

Linha 12 : Atualiza-se o valor da variável low_i para o maior valor entre low_i e o parâmetro $value_i.x$, visto que v_i é um nó maximizador (note que o valor de low_i somente será alterado se $value_i.x > low_i$).

Linha 13 : Atualiza-se o valor da variável $value_i$ para o maior valor entre $value_i$ e o parâmetro $value_i.x$, uma vez que por v_i se tratar de um nó de maximização ele deve possuir o maior valor produzido por um sucessor.

Linhas 14 e 15 : É verificado se a variável *cutoff* possui valor verdadeiro. Caso positivo, é chamada a rotina *send_cutoff* (descrita no pseudocódigo 9) que deve abortar todo o processamento dos escravos do processador P_k que avalia o nó v_i da variação corrente. Isto é feito porque os resultados de suas avaliações não teriam influência no resultado do processamento do mestre (como houve poda, não há estreitamento de janela).

Linhas 16 e 17 : Caso a condição da linha 14 não seja satisfeita, ou seja, não ocorreu poda, é verificado se a variável *improvement* contém valor 'true'.

Linha 18 : Se a condição da linha 17 for satisfeita é chamada a rotina *send_newbound(i)*, descrita no pseudocódigo 10, que deverá propagar as informações da nova janela de busca a todos os escravos de v_i. Tal comunicação tem o intuito de evitar o processamento desnecessário dos nós decorrente de uma janela desatualizada.

Linha 19 : Realiza a atualização da janela pela função *update_window_max*, descrita no pseudocódigo 11, visto que trata-se de um nó de maximização.

Linha 23 : É verificado se v_i é um nó minimizador, caso afirmativo, executam-se os procedimentos descritos da linha 24 a 41.

Linhas 24 e 25 : Repetem-se os mesmos procedimentos descritos nas linhas 4 e 5, respectivamente.

Linhas 26 e 27 : É verificado se o valor $valor_i.x$ da avaliação do nó $v_i.x$ retornado por P_k é menor ou igual a $alpha_i$ (janela inferior repassada inicialmente ao nó enviado a P_k). Caso positivo é identificada uma poda alfa e o valor da variável *cutoff* é modificado para 'true'.

Linhas 29 e 30 : É verificado se o valor $valor_i.x$ do nó avaliado por P_k é menor que o valor do nível superior da janela de busca, $high_i$ do nó v_i da variação corrente. Caso positivo é identificado um estreitamento no limite superior da janela alfa-beta de v_i assinalado atribuindo-se o valor 'true' à variável *improvement*.

Linha 32 : Atualiza-se o valor da variável $high_i$ para o menor valor entre $high_i$ e o parâmetro $value_i.x$, visto que v_i é um nó minimizador (note que o valor de $high_i$ somente será alterado se $value_i.x < high_i$).

Linha 33 : Atualiza-se o valor da variável $value_i$ para o menor valor entre $value_i$ e o parâmetro $value_i.x$, uma vez que por v_i se tratar de nó de minimização ele deve possuir o menor valor produzido por um sucessor.

Linhas 34 a 38 : Repetem-se os mesmos procedimentos das linhas de 14 a 18.

Linha 39 : Realiza a atualização da janela pela função *update_window_min*, descrita no pseudocódigo 12, visto que trata-se de um nó de minimização.

O pseudocódigo 9 descreve o procedimento de poda *send_cutoff*, enviado por P_j, que aborta a execução de todos os seus *slaves* do nível i (passado como parâmetro) ao nível t, onde i representa a raiz da variação corrente do processador P_j e t corresponde ao nível atual da variação corrente de P_j. Tal mensagem resulta de uma poda ocorrida no processamento do nó $v_i.x$ avaliado pelo escravo P_k de P_j. Observe na figura 6.5 que a variação corrente do nó v_i pode estar em uma profundidade t maior do que a profundidade i - nível em que é tratado o resultado enviado por P_k referente ao nó $v_i.x$. Desta forma, podas devem ser tratadas nos níveis intermediários entre i e t, visto que, os resultados das avaliações compreendidas neste intervalo não influenciariam no resultado do processamento de v_i. Na figura 6.5 a linha pontilhada ilustra o caso de poda propagado pelo no v_i. Note que a *linha 6* atualiza a lista de escravos de P_j, tornando-a vazia em função das podas.

Pseudocódigo 9 Rotina que envia mensagens Cutoff

```
1: def send_cutoff(int:i)
2: for q = i to t do
3:    for s in slaves_q do
4:       envia mensagem cutoff para 'q'
5:    end for
6:    slaves_l = {}
7: end for
```

O procedimento *send_newbound*, mostrado no pseudocódigo 10, comunica a todos os slaves do nível i (recebido como parâmetro) a nova janela de busca. A atualização da janela também deve ser propagada até o nível t do processador da variação corrente do nó v_i, bem como até o nível t do processamento de seus escravos. Obviamente, tal atualização leva em conta o fato de os

nós receptores de uma mensagem serem minimizadores ou maximizadores. O pseudocódigo 11 descreve a função de propagação da nova janela de busca para o caso de se tratar de um nó de maximização, ao passo que o pseudocódigo 12 efetua a mesma tarefa, porém, para um nó de minimização.

Pseudocódigo 10 Rotina que envia mensagens Newbound

```
1: def send_newbound(i)
2: for s in slaves_i do
3:     envia mensagem newbound para 's'
4: end for
```

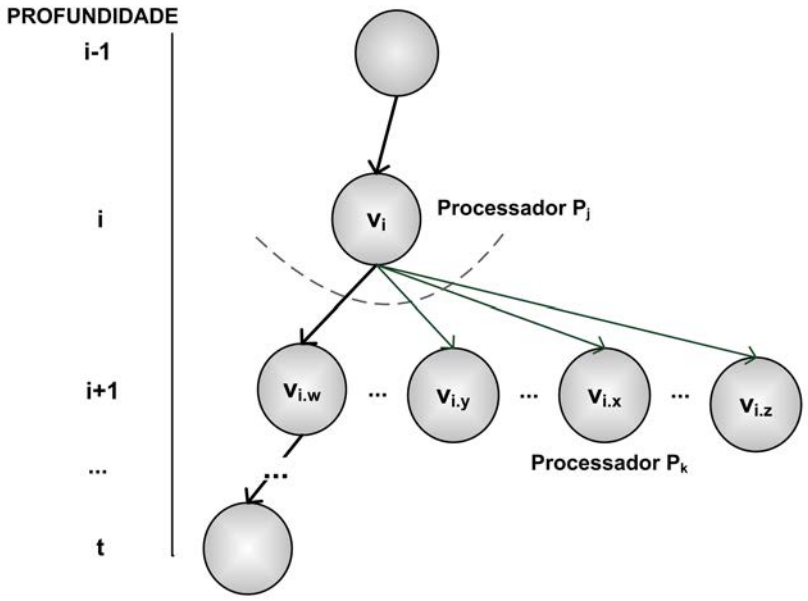

FIGURA 6.5: Exemplo da expansão de uma árvore de jogo pelo D-MA-Draughts destacando a variação corrente do processador P_j

Os procedimentos descritos no Pseudocódigo 11 são descritos abaixo de forma detalhada.

Linha 1 Definição da função *update_window_max* que recebe como parâmetro a profundidade *depth* a partir de onde será efetuado o processo de atualização da janela de busca e o valor *alpha* correspondente ao limite inferior corrente da janela de busca do mestre P_j.

Linhas 2 e 3 : Declaração das variáveis do tipo booleano *cutoff* e *improvement* que verificam se há uma poda ou estreitamento da janela alfa-beta, respectivamente;

Linha 4 : Executa os procedimentos contidos da linha 5 a 20 de modo que d seja igual a t, onde t é o nível atual da variação corrente.

Linha 5 : Verifica se o valor de *alpha* no nível *depth* ($alpha_{depth}$) é menor que o parâmetro *alpha*. A variável *improvement* recebe o valor correspondente a esta comparação.

Linha 6 : O valor de $alpha_{depth}$ é modificado para o maior valor entre $alpha_{depth}$ e o parâmetro *alpha*.

Pseudocódigo 11 Rotina que trata atualização de janela enviada por pai maximizador

```
1: def update_window_max(int:depth, double:alpha)
2: cuttof = false
3: improvement = false
4: for each detph to t do
5:    improvement = alpha_depth < alpha
6:    alpha_depth = max(alpha_depth, alpha)
7:    if v_depth é um max-node then
8:       low_depth = max(low_depth, alpha)
9:    end if
10:   if v_depth é um min-node then
11:      if high_depth ≤ alpha then
12:         cutoff = true
13:      end if
14:   end if
15:   if improvement and not cutoff then
16:      send_newbound(depth)
17:   end if
18:   if (not improvement) or (cutoff) then
19:      break;
20:   end if
21: end for
22: if cutoff then
23:    send_cutoff(depth)
24: end if
```

Linha 7 e 8 : É verificado se o nó em questão, v_{detph}, trata-se de um nó de maximização. Caso positivo, a variável que corresponde ao limite inferior da janela alfa-beta do nível *depth*, low_{depth}, é atualizada para o maior valor entre low_{depth} e o parâmetro *alpha*.

Linha 10 a 12 : É verificado se o nó em questão, v_{depth}, é um nó de minimização. Caso positivo, é verificado se o limite superior da janela alfa-beta do nível *depth*, $high_{depth}$, é menor ou igual ao parâmetro *alpha*. Se esta última condição for satisfeita, é atribuído o valor '*true*' à variável *cutoff*, fato que identifica que o processamento dos filhos de v_{depth} deve ser abortado.

Linhas 15 e 16 : Se foi identificado um estreitamento de janela (*improvement* possui valor '*true*') e nenhuma poda foi detectada (*cutoff* possui valor '*false*'), o procedimento *send_newbound* (descrito no pseudocódigo 10) é chamado para repassar os novos limites aos filhos de v_{depth}.

Linhas 18 e 19 : Se não foi identificado um estreitamento de janela (*improvement* possui valor 'false'), o laço é imediatamente interrompido, pois não há informações de alterações de limites da janela de busca a ser transmitido Neste caso, o processamento dos filhos de v_{depth} prosseguirá normalmente, sem alterações. O mesmo ocorrerá no caso de uma poda ter sido detectada, já que, neste caso, todo o processamento abaixo do nível *depth* deverá ser abortado.

Linhas 22 e 23 Se ao final do laço a variável *cutoff* identificar que houve uma poda (*cutoff* possui valor 'true'), o procedimento *send_cutoff* (descrito no pseudocódigo 9) é chamado para efetivamente abortar todo o processamento dos filhos de v_{depth}.

Pseudocódigo 12 Rotina que trata atualização de janela no nível de minimização

```
1: def update_window_min(int:depth, double:beta)
2: cuttof = false
3: improvement = false
4: for each depth to t do
5:    improvement = beta_depth > beta
6:    beta_detph = min(beta_depth, beta)
7:    if v_depth is a min-node then
8:       high_depth = min(high_depth, beta)
9:    end if
10:   if v_depth is a max-node then
11:      if low_depth ≥ beta then
12:         cutoff = true
13:      end if
14:   end if
15:   if improvement and not cutoff then
16:      send_newbound(depth)
17:   end if
18:   if (not improvement) or (cutoff) then
19:      break;
20:   end if
21: end for
22: if cutoff then
23:    send_cutoff(depth)
24: end if
```

Os procedimentos contidos no pseudocódigo 12 são semelhantes aos do pseudocódigo 11, salvo por agora se tratar de um nó do tipo minimizador, por isso, serão descritas apenas as linhas do pseudocódigo que destacam esta diferença.

Linha 1 : Definição da função *update_window_min* que recebe como parâmetro a profundidade *depth* a partir de onde será efetuado o processo de atualização da janela de busca e o valor *beta* do limite superior corrente da janela de busca do mestre P_j.

Linha 5 : Verifica se o valor de beta no nível *depth* ($beta_{detph}$) é maior que o parâmetro beta. A variável *improvement* recebe o valor correspondente a esta comparação.

Linha 6 : O valor de $beta_{depth}$ é modificado para o menor valor entre $beta_{detph}$ e o parâmetro beta.

Linha 7 e 8 : É verificado se o nó em questão, v_{depth}, trata-se de um nó de minimização. Caso positivo, a variável que corresponde ao limite superior da janela alfa-beta do nível *detph*, $high_{depth}$, é atualizada para o menor valor entre $high_{depth}$ e o parâmetro *beta*.

Linha 10 a 12 : É verificado se o nó em questão, v_{depth}, é um nó de maximização. Caso positivo, é verificado se o limite inferior da janela alfa-beta do nível *depth*, low_{depth}, é maior ou igual ao parâmetro *beta*. Se esta última condição for satisfeita, é atribuído o valor 'true' à variável *cutoff*, fato que identifica uma poda.

A próxima seção apresenta um exemplo da dinâmica de atuação do YBWC no *D-MA-Draughts*.

6.2.2 Exemplo de atuação do YBWC no D-MA-Draughts

Sempre que o *D-MA-Draughts* inicia uma busca, o estado corrente N_i é colocado na pilha local de P_0 (*Processador Principal*) e as pilhas dos *Processadores Auxiliares* estarão vazias. O sistema então inicia a expansão serial da sub-árvore mais à esquerda de N_i na profundidade máxima e, ao concluí-la, será capaz de distribuir a expansão das demais subárvores de N_i. Quando é enviada alguma requisição (*RequestTask*) de um processador P_k para P_0, P_0 verificará, dentre os nós que ainda não foram explorados em sua pilha local, algum disponível. O *D-MA-Draughts* escolhe, dentre os nós disponíveis, o mais profundo e mais à esquerda não explorado da pilha local de P_0. Tal escolha se deve ao fato de esta estratégia permitir uma conclusão mais rápida da sub-árvore corrente de P_0, uma vez que agiliza o processo de estreitamento de janelas.

A figura 6.6 ilustra uma árvore de jogo e sua respectiva pilha de estados a fim de exemplificar a dinâmica de atuação do *D-MA-Draughts* descrita acima, bem como a aplicação das mensagens utilizadas para comunicação entre processos apresentadas neste capítulo.

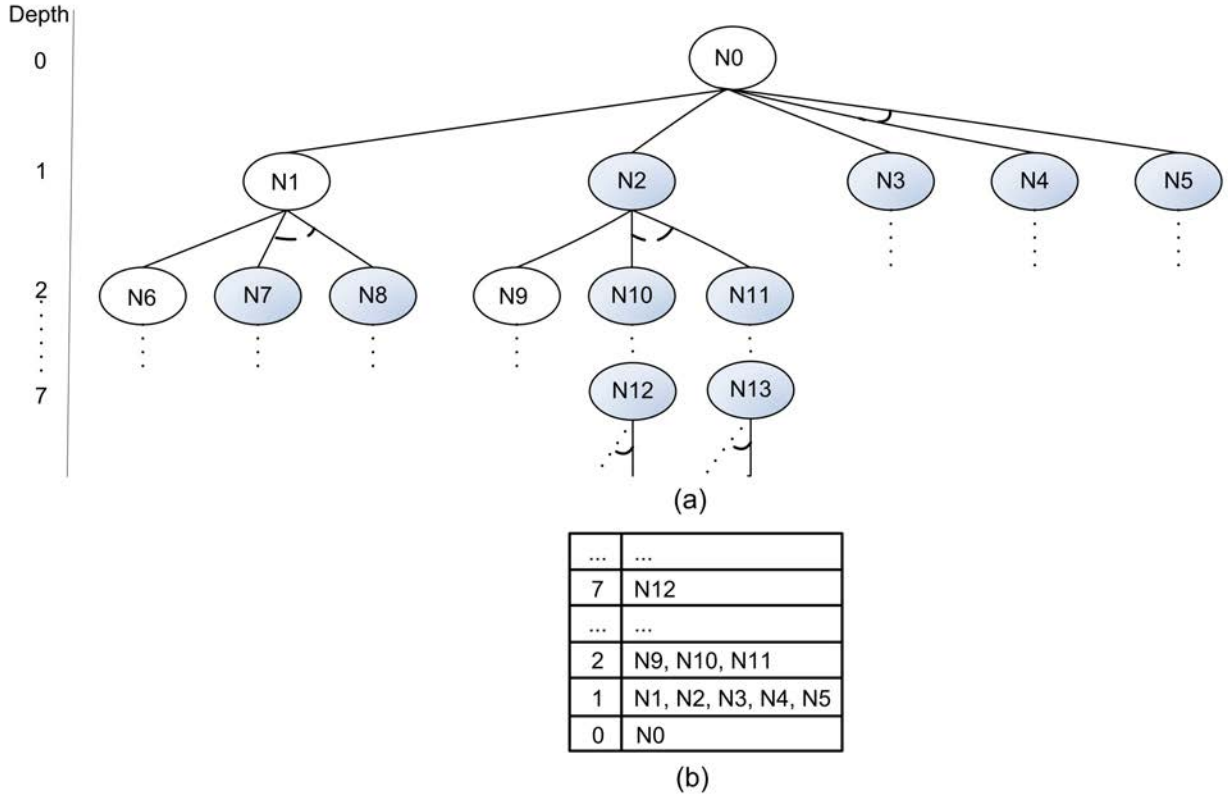

FIGURA 6.6: (a) Árvore de exemplo (b) Ilustração da pilha de expansão do nó N_{12}

Na Figura 6.6, N_0 representa o estado atual do tabuleiro em um dado momento de uma partida e é repassado ao módulo de busca que será explorado por P_0. Quando um processador P_k envia uma *RequestTask* para P_0, as seguintes situações podem ser dadas como exemplos:

1. se P_0 está explorando N_8, ele informa que não há tarefa disponível;

2. se P_0 está explorando N_{12}, ou seja, os nós disponíveis são: N_3, N_4, N_5 e N_{11}, ele envia o nó N_{11} seguido da mensagem *ResponseTask*;

3. se N_{11} já tiver sido distribuído (neste caso estarão disponíveis N_3, N_4 e N_5), ele envia N_3.

Ressalta-se que os nós destacados na Figura 6.6 representam os nós que podem ser distribuídos.

Considere agora a situação em que P_0 está explorando o nó N_{10}. Se um processador P_k envia uma solicitação de tarefa para P_0 este último enviará N_{11}. Todavia, se durante a exploração de N_{10} o algoritmo de busca detectar que a sub-árvore N_{11} pode ser cortada, P_0 deverá enviar uma mensagem *Cutoff* a todos os processadores, incluindo P_k. Neste caso, se P_k já tiver iniciado a exploração de N_{11}, ele irá abortar suas atividades imediatamente, caso contrário, ele nem iniciará o processo de expansão. Além disso, como o algoritmo Alfa-Beta trabalha com uma janela contendo os limites de minimização e maximização que podem acarretar em uma poda, sempre que um processador mestre detectar uma redução dos limites da janela alfa-beta, ele deve transmitir estes novos limites para todos os escravos por meio da mensagem *Newbound*. Por exemplo, na mesma situação descrita no exemplo anterior, se na expansão serial da subárvore completa de N_{10} por P_0 ocorre um estreitamento de limites da janela alfa-beta, P_0 deve enviar uma mensagem *Newbound* para todos os escravos (incluindo P_k), comunicando os novos limites da janela de busca. Caso a mensagem não fosse enviada, P_k poderia expandir N_{11} com uma janela alfa-beta larga (antiga) o que poderia desperdiçar uma eventual possibilidade de poda na expansão de N_{11}.

A tabela 6.1 resume as mensagens que compõem o protocolo de comunicação do *D-MA-Draughts*. Na seção 6.2.5 será apresentada a interface de comunicação utilizada pelo *D-MA-Draughts* que permite as trocas de mensagens entre os processos.

Como o YBWC do *D-MA-Draughts* utiliza TT e aprofundamento iterativo a fim de melhorar a eficiência de seu módulo de busca, as seções 6.2.3 e 6.2.4 apresentam como tais técnicas foram empregadas neste sistema.

Tabela 6.1: Mensagens do protocolo de comunicação do D-MA-Draughts

Tipo	Descrição
InitSearch	Mensagem enviada por P_0 a todos os *Processadores Auxiliares* indicando o início da busca. Assim, os *Processadores Auxiliares* deixarão o estado *inativo* passando para o estado *ocioso*, onde iniciarão sua procura por tarefa. Em um jogo de treinamento da rede neural, uma cópia da rede neural deve ser enviada para os *Processadores Auxiliares* para que estes atualizem os pesos de suas redes.
EndSearch	P_0 envia tal mensagem a todos os *Processadores Auxiliares* a fim de instruí-los a pararem de solicitar tarefas e voltarem ao estado *inativo*
RequestTask	Utilizada por processadores ociosos para solicitar tarefa a outro processador aleatoriamente
ResponseTask	Resposta a 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	Mensagem utilizada por processadores *slaves* para retornar o resultado de sua avaliação ao 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	Mensagem enviada por P_0 a todos os *Processadores Auxiliares* para que estes finalizem suas execuções (fim do programa)

6.2.3 Tabela de Transposição

Conforme previsto na seção 2.9, o *D-MA-Draughts* utiliza a versão *fail-soft* do algoritmo Alfa-Beta para implementação do YBWC, visto que esta versão permite a integração com TT. O *D-MA-Draughts* utiliza TT para armazenar estados de tabuleiro que foram previamente submetidos ao procedimento de busca [68]. Desta forma, se um estado S for submetido ao procedimento de busca, primeiramente, será verificado se o mesmo se encontra armazenado na TT. Caso positivo, se os valores da TT satisfazem um dado conjunto de restrições (a ser apresentado na seção 6.2.3.6), o estado armazenado será utilizado, evitando o processamento desnecessário. Caso negativo, o estado S será avaliado e armazenado na TT, conforme será apresentado na seção 6.2.3.5.

A TT foi implementada no *D-MA-Draughts* como uma tabela *hash*. Uma tabela *hash* é uma estrutura de dados que associa *chaves* a *valores* [30]. As chaves são usadas para representar os estados do tabuleiro do jogo de damas e são obtidas a partir de informações relevantes referentes àqueles estados. Para construção destas chaves o *D-MA-Draughts* utiliza a técnica de Zobrist [73], detalhada na próxima seção.

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

O método descrito por Zobrist [73] 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", caso contrário, a operação resulta em "falso". Computacionalmente, o operador **XOR** pode ser aplicado sobre dois operandos numéricos [18]. Neste caso, os operadores "verdadeiro" e "falso" correspondem, respectivamente, aos bits "1" e "0". 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 $Seq_1 = b_1, b_2, ..., b_n$ uma sequência binária de n bits. Além disso,considere $Seq_2 = r_1, r_2, ..., r_n$ 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 $Seq_3 = b_1 \otimes r_1, b_2 \otimes r_2, ..., b_n \otimes rn$;

2. operandos numéricos na base decimal: a operação **XOR** sobre dois inteiros decimais segue o mesmo procedimento mostrado para operandos numéricos na base binária, sendo que, neste caso, os dois argumentos inteiros decimais devem ser, antes de tudo, convertidos para a base binária. A conversão de inteiros decimais para binários é transparente em C++ [1]. Isso significa que dois operandos inteiros decimais podem ser passados como argumentos para o operador **XOR** (a conversão é feita implicitamente).

O operador **XOR** aplicado sobre sequências aleatórias (r) de inteiros decimais de n bits respeita as seguintes propriedades [73]:

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 $s_i = r_1 \otimes r_2 \otimes ... \otimes r_i$ 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).

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

[1] linguagem usada na implementação do jogador

qualquer elemento for adicionado ou retirado do subconjunto S_0, a chave *hash* mudará pelo inteiro que corresponde àquele elemento. No caso do *D-MA-Draughts*, existem 2 tipos distintos de peças (peça simples e rei), 2 cores distintas de peças (peça preta e peça vermelha) e 32 casas no tabuleiro do jogo. Então, existem, no máximo, 128 possibilidades distintas (2x2x32) de colocar alguma peça em alguma casa do tabuleiro. Assim, criou-se um vetor de 128 elementos inteiros aleatórios, mostrado na figura 6.7, 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 6.7, 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.

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 6.7: Vetor de 128 elementos inteiros aleatórios utilizados pelo D-MA-Draughts

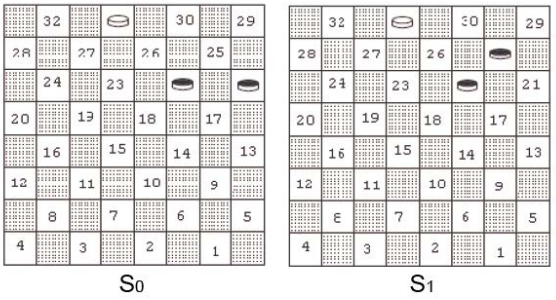

FIGURA 6.8: Exemplo de movimento simples.

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 [73]. Isto devido à 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 de 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 6.8 e o vetor de números aleatórios utilizados pelo *D-MA-Draughts*, mostrado na figura 6.7:

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 $I_{21} = 1171196056361380757$;
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, $C_1 = 8988798927364941823$.

6.2.3.2 Estrutura ENTRY - Dados Armazenados para um Determinado Estado do Tabuleiro do Jogo

A TT utilizada pelo *D-MA-Draughts* (implementada como uma tabela *hash* com a finalidade de obter máxima velocidade de manipulação) mantém um registro de estados analisados do tabuleiro do jogo com os seus respectivos valores de avaliação (predição) obtidos pelo algoritmo de busca. A partir de então, sempre que um estado do tabuleiro for apresentado como entrada do procedimento de busca, primeiro o algoritmo verificará se esse estado se encontra na TT e, caso afirmativo, verifica se os valores armazenados na TT satisfazem um dado conjunto de restrições a ser definido na seção 6.2.3.6. Caso ambas as condições sejam atendidas o sistema utilizará os valores armazenados em memória, o que abreviará o retorno da busca. Caso negativo, a busca prosseguirá sem alteração.

O *D-MA-Draughts* utiliza uma estrutura chamada ENTRY para armazenar na TT os dados de entrada relativos a um determinado estado S avaliado pelo YBWC, conforme disposto a seguir.

Pseudocódigo 13 Estrutura ENTRY relativa a um estado S

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

No pseudocódigo 13, o campo *hashvalue* armazena a chave *hash* correspondente a S; o campo *bestvalue* armazena o valor da predição de S retornada pelo algoritmo de busca (YBWC); o campo *bestmove* armazena a melhor jogada a partir de S, que é, também, retornada pelo algoritmo de busca; o campo *depth* armazena a profundidade de busca no momento da avaliação de S; 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. Isso será melhor explicado na seção 6.2.3.5. 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. Conforme será visto a seguir, esta segunda chave tem como objetivo atacar o problema de colisão.

A estrutura ENTRY é a unidade básica de entrada para a TT utilizada pelo *D-MA-Draughts*. O pseudocódigo 14 apresenta um exemplo de armazenamento de dados em uma estrutura *ENTRY* na TT.

Pseudocódigo 14 Exemplo de armazenamento na estrutura ENTRY de um estado *S*

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

A seção 6.2.3.3 abaixo explica os possíveis problemas de colisão que podem ocorrer na TT e como a segunda chave *hash*, no caso *checksum*, pode ser usada como um dos instrumentos para detecção e tratamento de colisões.

6.2.3.3 Colisões - Conflitos de Endereços para Estados do Tabuleiro do Jogo

A TT do *D-MA-Draughts* trata dois tipos de erros identificados por Zobrist [73]. 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 ao consultar a TT.

Para controlar os erros *tipo 1*, são usadas as duas chaves *hash* da estrutura ENTRY : *hashvalue* e *checksum*. Estas chaves são geradas 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, *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 TT. Para resolver este problema, o *D-MA-Draughts* utiliza dois esquemas de substituição, um chamado *Deep* e outro chamado *New*. Esses esquemas foram escolhidos entre os sete estudados por Breuker em [74], [75]. A escolha desses dois esquemas foi feita baseada no jogador *Chinook* [25], [4]. O *Chinook* utiliza uma TT de dois níveis em que cada entrada da

tabela (endereço) pode conter informações relativas a até 2 estados do tabuleiro (um estado em cada nível). Experimentos no *Chinook* mostram que tal estrutura para a TT reduz o tamanho da árvore de busca em até 10% [72].

Caso um endereço da TT 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 TT 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.

Assim sendo, a tabela *TTABLE* contém 2 vetores *e1* e *e2*, do tipo *ENTRY*, representando os 2 níveis da TT, e cada esquema de substituição está associado a um dos dois vetores de *TTABLE*, como mostrado abaixo:

- ***Deep***: o esquema de substituição *Deep* armazena no primeiro vetor de *TTABLE* (*ENTRY* e1*) o estado com a mais profunda sub-árvore, ou seja, se em uma posição da TT já houver armazenado um estado igual ao estado corrente a ser armazenado, será mantido na TT o estado com maior profundidade. 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 de busca 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 TT *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 *D-MA-Draughts* [18].

A seguir é apresentada a estrutura da TT bem como o modo como os esquemas *deep* e *new* são usados para resolver colisões.

6.2.3.4 Estrutura TTABLE - Manipulação de Dados na Tabela de Transposição com Tratamento de Colisões

A TT utilizada pelo *D-MA-Draughts* é uma estrutura do tipo *TTABLE* composta por dois vetores, (*e1* e *e2* com elementos do tipo *ENTRY*). Além disso, existe um método para armazenar novas entradas e um método para ler as entradas já existentes na tabela.

O pseudocódigo 15 apresenta a estrutura de *TTABLE*.

Pseudocódigo 15 Estrutura de *TTABLE*

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

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

O campo *tableSize* define o espaço alocado na TT para manipulação de registros do tipo *ENTRY* na memória. É importante notar que *TTABLE* utiliza tanta memória quanto houver disponível e, claro, quanto mais memória, melhor o desempenho da TT.

Cada par de vetor e_1 e e_2 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 pelos métodos *StoreEntry(...)* e *GetEntry(...)* apresentados nas expressões 6.1 e 6.2, respectivamente.

$$StoreEntry(newEntry); \tag{6.1}$$

O método *StoreEntry(newEntry)*, apresentado na expressão 6.1, representa o método de armazenamento de informações sobre um dado estado do tabuleiro na TT. 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*: devido à inexistência de 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 gravando o valor de *newEntry* em e_2 independentemente do conteúdo do segundo vetor e_2;

$$GetEntry(n; hashvalue; checksum; pdepth); \tag{6.2}$$

O método *GetEntry*(...) da expressão 6.2 visa recuperar informações sobre um dado estado do tabuleiro eventualmente armazenado na TT. Para tanto, ele recebe 4 parâmetros de entrada associados ao estado: 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 (que coincide com a profundidade corrente de busca).

A partir daí, *GetEntry* localiza o endereço E_1 associado ao parâmetro *hashvalue* na TT 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 positivo, 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 de busca pode utilizar os dados armazenados em memória ao invés 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.

Em resumo, havendo uma TT como a *TTABLE*, sempre que um determinado estado do tabuleiro for apresentado ao algoritmo de busca, ele verificará, primeiro, a TT para ver se aquele estado do tabuleiro já foi analisado. Caso afirmativo e o conjunto de restrições (apresentado na seção 6.2.3.6) também seja satisfeito, 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 TT.

6.2.3.5 Armazenamento dos Estados do Tabuleiro na Tabela de Transposição a partir do Algoritmo YBWC

A estrutura *ENTRY*, mostrada na seção 6.2.3.2, é a chave básica de entrada para a TT. Desta forma, para criação de uma estrutura do tipo *ENTRY* o *D-MA-Draughts* utiliza o método *store* cuja assinatura é representada na expressão 6.4:

$$store(n;\ besteval;\ bestmove;\ depth;\ scoreType); \tag{6.3}$$

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 6.2.3.1), o método acima já possui todas as informações para escrever na TT, bastando obedecer o esquema de substituição descrito na seção 6.2.3.4. Assim sendo, o método *store* invoca o método *StoreEntry* (expressão 6.1) e conclui o armazenamento do nó avaliado na TT.

O detalhe mais importante do método *store* é o parâmetro *scoreType*. Ele pode assumir os valores *hashExact*, *hashAtLeast* ou *hashAtMost* de acordo com a 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 TT 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 $P_1, ..., P_k$, então $P_i \epsilon \{P_1, ..., P_k\}$, isto é, P_i é a predição do filho F_i de n mais à esquerda, onde tal predição satisfaz a seguinte expressão:

     $$P_i < \alpha; \tag{6.4}$$

     sendo *alpha* o limite inferior do intervalo de busca repassado por n para a avaliação de F_i (os irmãos de F_i à sua direita foram podados durante a avaliação de 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 $P_1, ..., P_k$, então $P_i \epsilon \{P_1, ..., P_k\}$, isto é, P_i é a predição do filhos F_i de n mais à esquerda, onde tal predição satisfaz a seguinte expressão;
   - $P_i > \beta$

- sendo beta o limite superior do intervalo de busca repassado por n para a avaliação de F_i (os irmão de F_i à sua direita foram podados durante a avaliação de n).

A partir deste momento, quando forem encontradas as expressões *hashExact*, *hashAtMost* ou *hashAtLeast* associadas às predições dos estados de tabuleiros, considere que essas predições foram obtidas seguindo os critérios apresentados acima.

6.2.3.6 Recuperação dos Estados do Tabuleiro da Tabela de Transposição a partir do Algoritmo YBWC

O *D-MA-Draughts* tenta, sempre que possível, utilizar os dados armazenados em memória em vez de expandir uma árvore de busca do jogo. O método para verificar se um determinado estado do tabuleiro do jogo n encontra-se armazenado na TT (memória) possui a seguinte assinatura:

$$retrieve(n;\ besteval;\ bestmove;\ pdepth;\ nodeType); \tag{6.5}$$

O estado do tabuleiro do jogo que está sendo procurado na TT é representado por n; *pdepth* 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 TT.

Assim sendo, inicialmente, o método *retrieve* aciona o procedimento de leitura *GetEntry* (6.2) para checar se o estado n está armazenado na TT. 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 TT 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 valor do parâmetro *nodeType* representa o fato de o pai de n ser maximizador ou minimizador.

A seguir apresentam-se as restrições que precisam ser satisfeitas para que os valores da TT referentes a nós minimizadores ou maximizadores possam ser utilizados pelo YBWC. Considere a árvore da figura 6.9 que ilustra um caso de busca em que os nós minimizadores S_1 e S_2 (com pais maximizadores) foram armazenados na TT conforme descrito a seguir (pode-se notar que S_2 tem k filhos e que houve uma poda alfa a partir do filho S_{i+1}):

Note que S_1 está na TT com valor de predição P_1 e *scoreType* igual a *hashExact*. S_2 está na TT com um valor de predição *hashAtMost* $P_i^{'}$. Além disso, ambos foram armazenados com profundidade *depth*;

Neste caso, se os estados S_1 e S_2 ocorrerem novamente por transposição com profundidade $depth_1$, seus valores na TT somente poderão ser utilizadas se as seguintes restrições forem atendidas:

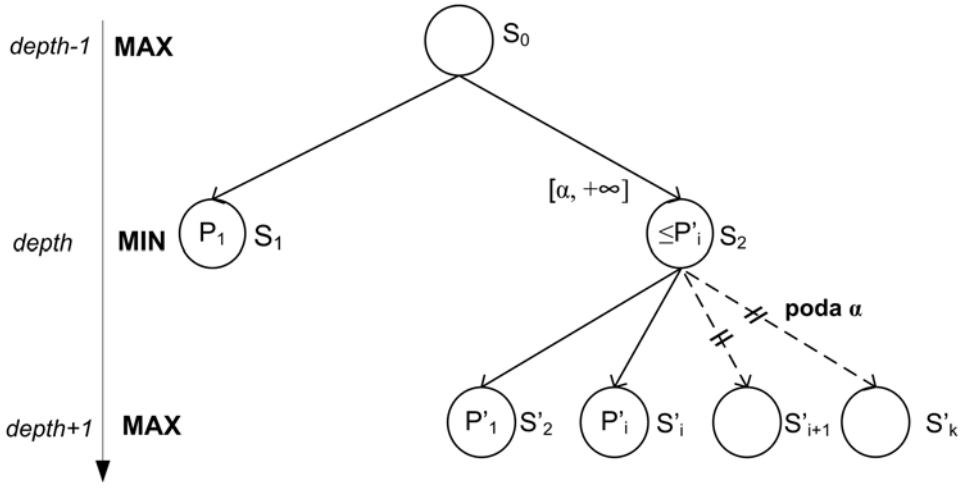

FIGURA 6.9: Árvore de busca onde o pai de S_1 e S_2 é maximizador

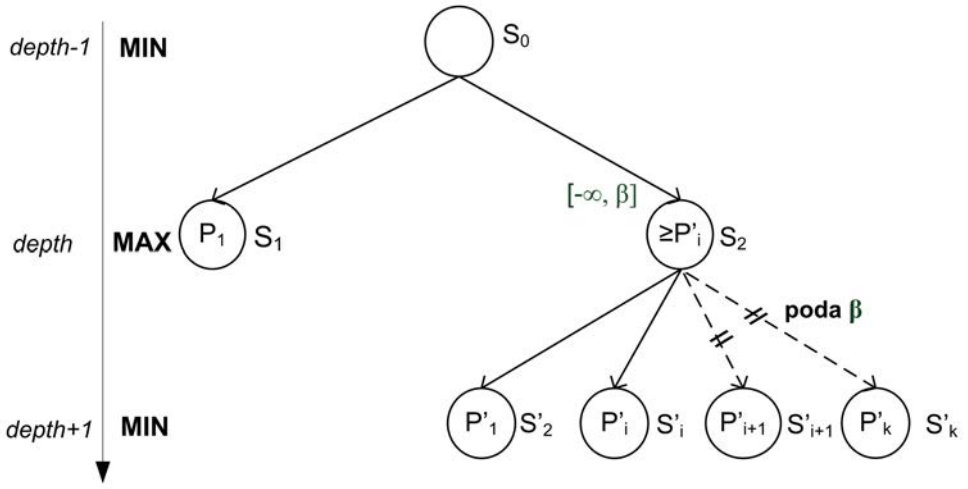

FIGURA 6.10: Árvore de busca onde o pai de S_1 e S_2 é minimizador

1. o valor de P_1 pode ser usado desde que o valor *scoreType* seja igual a *hashExact*, ou seja, P_1 possui o mesmo nível de S_1 (nível de minimização) e $depth_1 \geq depth$;

2. $P_i^{'}$ somente pode ser usado se $depth_1 \geq depth$ e $P_i^{'} \leq \alpha$, onde α é o limite inferior do intervalo de busca com que S_2 seria expandido. Note que, S_2 nunca seria escolhido pelo pai S_0, visto que nenhum de seus filhos com predição menor que $P_i^{'}$ "subiria" para S_0, pois o valor de α impediria. Analogamente, nenhum filho de S_2 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 os nós S_1 e S_2 forem nós maximizadores (com pais minimizadores), considere a árvore da figura 6.10:

Partindo da premissa de que a profundidade *depth* 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 maximização, ou seja, no mesmo nível de S_2;
2. se $P_i^{'} \geq \beta$, o valor associado a $P_i^{'}$ pode ser usado, uma vez que a predição associada a S_2 será descartada, pois:
   - todo irmão $S_k^{'}$ de $S_i^{'}$ que tiver sido podado na poda beta que gerou $S_i^{'}$, e que tiver predição $P_k^{'} > P_i^{'}$ "subirá" pelo maximizador de S_2, mas será barrado pelo minimizador de β;
   - todo irmão $S_k^{'}$ de $S_i^{'}$ que tiver sido podado na poda beta que gerou $S_i^{'}$, e que tiver predição $P_k^{'} < P_i^{'}$ será descartado pelo maximizador de S_2.

   É possível notar que se $P_i^{'} < \beta$, o valor associado a $P_i^{'}$ não pode ser usado, pois todo irmão $S_k^{'}$ de $S_i^{'}$ que tiver sido podado na poda beta que gerou $S_i^{'}$, e que tiver predição $P_k^{'}$, tal que $P_i^{'} < P_k^{'} < \beta$ seria candidato a "subir" até S_0. Inicialmente "subiria" até S_2 caso $P_k^{'}$ fosse maior que as predições dos filhos de S_2, posteriormente, "subiria" até S_0, caso $P_k^{'}$ fosse a menor entre as predições dos filhos de S_0;
3. caso nenhum dos itens acima possam ser usados, os estados S_1 e S_2 precisam ser expandidos.

6.2.4 Aprofundamento Iterativo

A técnica de aprofundamento iterativo contribui para o *look-ahead* do jogador, uma vez que permite a exploração de níveis mais profundos da árvore de jogo dentro de um determinado tempo (conforme apresentado na seção 2.5.2). O *D-MA-Draughts* utiliza a técnica de aprofundamento iterativo baseada na idéia de Atkin e procede da seguinte maneira: o algoritmo YBWC é chamado com profundidades *depth* = 8, 10, 12, ..., *max*, até que o tempo de busca se esgote em uma profundidade qualquer *depth* = *d*, tal que $8 \leq d \leq max$. Nessa 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).

A desvantagem do aprofundamento iterativo é o processamento repetido de estados de níveis mais rasos da árvore de busca. Para atenuar tal inconveniente, o *D-MA-Draughts* faz uso da TT cuja implementação foi descrita na seção 6.2.3. Para introduzir o aprofundamento iterativo no *D-MA-Draughts* é necessária uma rotina de iteração conforme a apresentada no pseudocódigo 16.

Esta rotina tem como parâmetros de entrada o nó n a ser avaliado; a profundidade máxima *depth* de busca; o valor *min* do limite inferior da janela de busca; o valor *max* do limite superior da janela de busca; e o parâmetro *bestvalue* que armazena a melhor ação a ser executada a partir de n, onde *besteval* também é o parâmetro de saída deste algoritmo. Note que na **linha 4** é realizado o laço de repetição desta rotina onde há o incremento da profundidade de busca em 2 unidades

Pseudocódigo 16 Rotina de iteração

```
1: iterative(node:n, int:depth, int:min, int:max, move:bestmove)
2: besteval = ybwc(n, 2, min, max, bestmove)
3: start = clock()
4: for d=4 ; d ≤ MAX-SEARCH-DEPTH ; d=d+2 do
5:    besteval = ybwc(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
```

até que se atinja a profundidade *depth* máxima passada como argumento. A **linha 5** invoca o procedimento de busca composto pelo algoritmo YBWC. O ponto mais importante desta rotina de iteração é apresentado nas **linhas 7** e **8** deste pseudocódigo, onde é avaliado se o tempo de execução excedeu o tempo máximo de busca definido na constante MAX-EXECUTION. Desta forma, se tal condição for satisfeita o algoritmo é interrompido e o valor da última avaliação é retornado. Caso negativo, a busca irá continuar com uma profundidade de busca acrescida.

6.2.5 Integração do D-MA-Draughts com MPI

A implementação do algoritmo YBWC no *D-MA-Draughts* foi baseado em troca de mensagens (resumidas na Tabela 6.1). Para isso, foi utilizado o mecanismo de comunicação entre processos chamado MPI (*Message Passing Interface*) [mpi]. A integração deste mecanismo com o *D-MA-Draughts* será apresentada através de diagramas que modelam a dinâmica do sistema - diagramas de estados [76] - devidamente comentados com as chamadas da MPI.

O primeiro diagrama (Figura 6.11) trata da inicialização e finalização da execução do *D-MA-Draughts* e é complementado por outros 3 diagramas de estados compostos (que especificam com detalhes alguns estados do sistema) [76]. Tais diagramas são:

1. diagrama *YBWC* (Figura 6.12), que trata dos estados de um processador que está executando a busca.
2. diagrama *ProcessaMensagensBusca* (Figura 6.13), que trata as mensagens recebidas por um processador durante a busca.
3. diagrama *TrataResultado* (Figura 6.14), que trata os valores de cada nó avaliado em um processo de busca, seja localmente ou remotamente, propagando novas janelas e podas.

No diagrama de estados principal, Figura 6.12, 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. Após esta chamada, a MPI atribui um número de identificação a cada processador, denominado *rank*. Para o *Processador Principal* é atribuído *rank* 0, que por sua vez, segue um caminho diferente dos *Processadores*

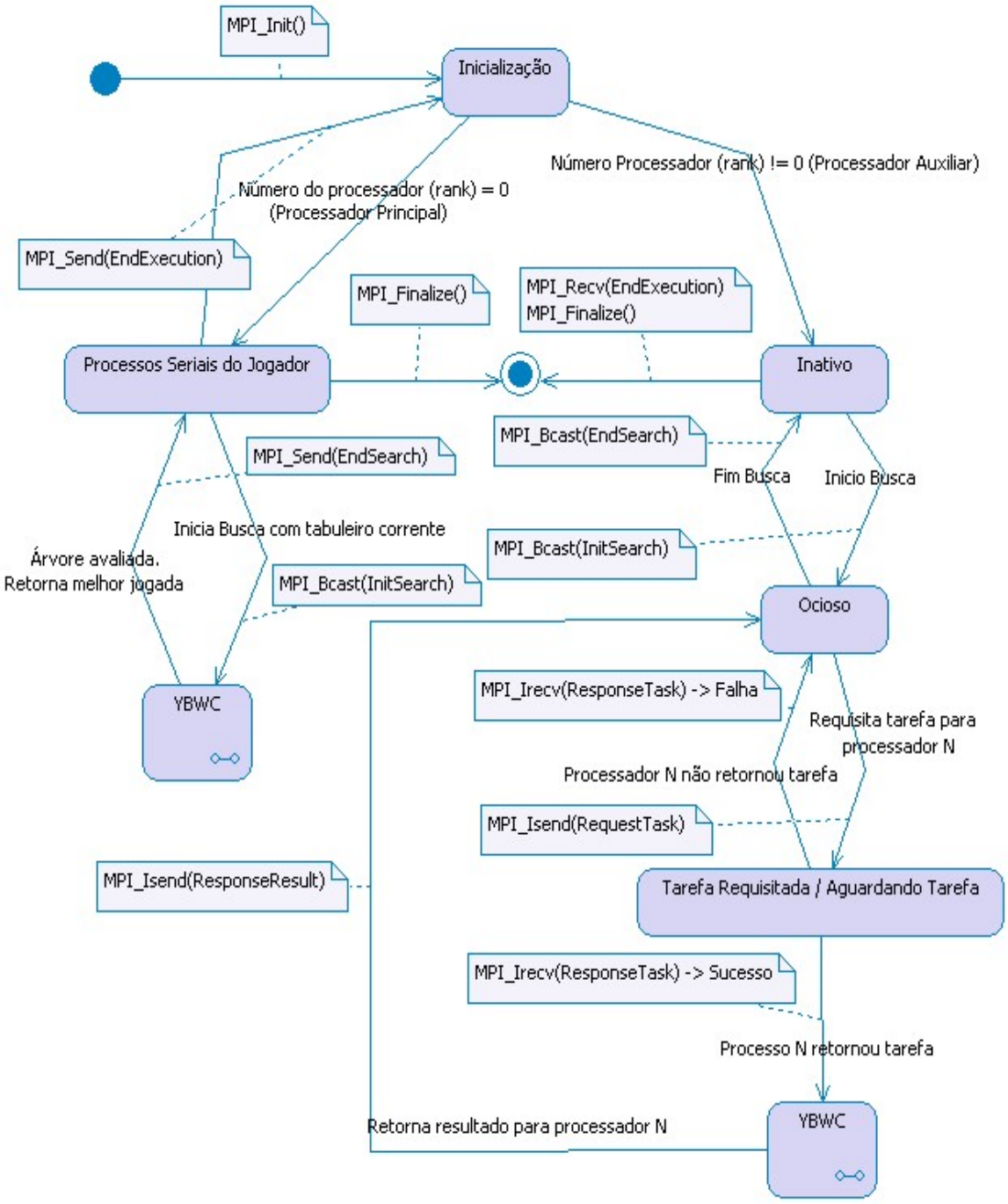

FIGURA 6.11: Diagrama de estados principal - início e fim dos processos

Auxiliares que, consequentemente, recebem um rank diferente de 0. Os *Processadores Auxiliares* entram em estado *inativo* aguardando a busca começar. O *Processador Principal*, como demonstrado na figura 6.3, fica encarregado de executar os processos seriais e, ao chegar na etapa de busca, emite uma mensagem *InitSearch* aos *Processadores Auxiliares*, que como descrito anteriormente, passam para o estado *ocioso* e começam a solicitar tarefas. O *Processador Principal*, após informar os demais processadores sobre o início da busca, vai para o estado *Busca AlphaBeta* representado no diagrama composto YBWC da figura 6.12 e começa a processar o estado corrente

como a raiz da árvore de busca. Os *Processadores Auxiliares* entram neste estado após obter sucesso na requisição de tarefa.

O algoritmo Alfa-Beta é executado com o subproblema recebido e, ao completar o processamento deste problema, 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 da Figura 6.14), 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 como tarefa outra parte dessa 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*, ela é tratada conforme mostrado no diagrama de estados composto *ProcessaMensagensBusca* mostrado na Figura 6.13.

O diagrama da Figura 6.13 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.
- *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 da Figura 6.14 apresenta o diagrama de estados composto *TrataResultado*. Aqui trata-se o resultado de um nó *vi.x* sucessor de um nó *vi* processado. Tal resultado pode causar tanto um estreitamento de janela quanto 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 *vi* e a profundidade em curso da variação corrente de v_i é abortada.

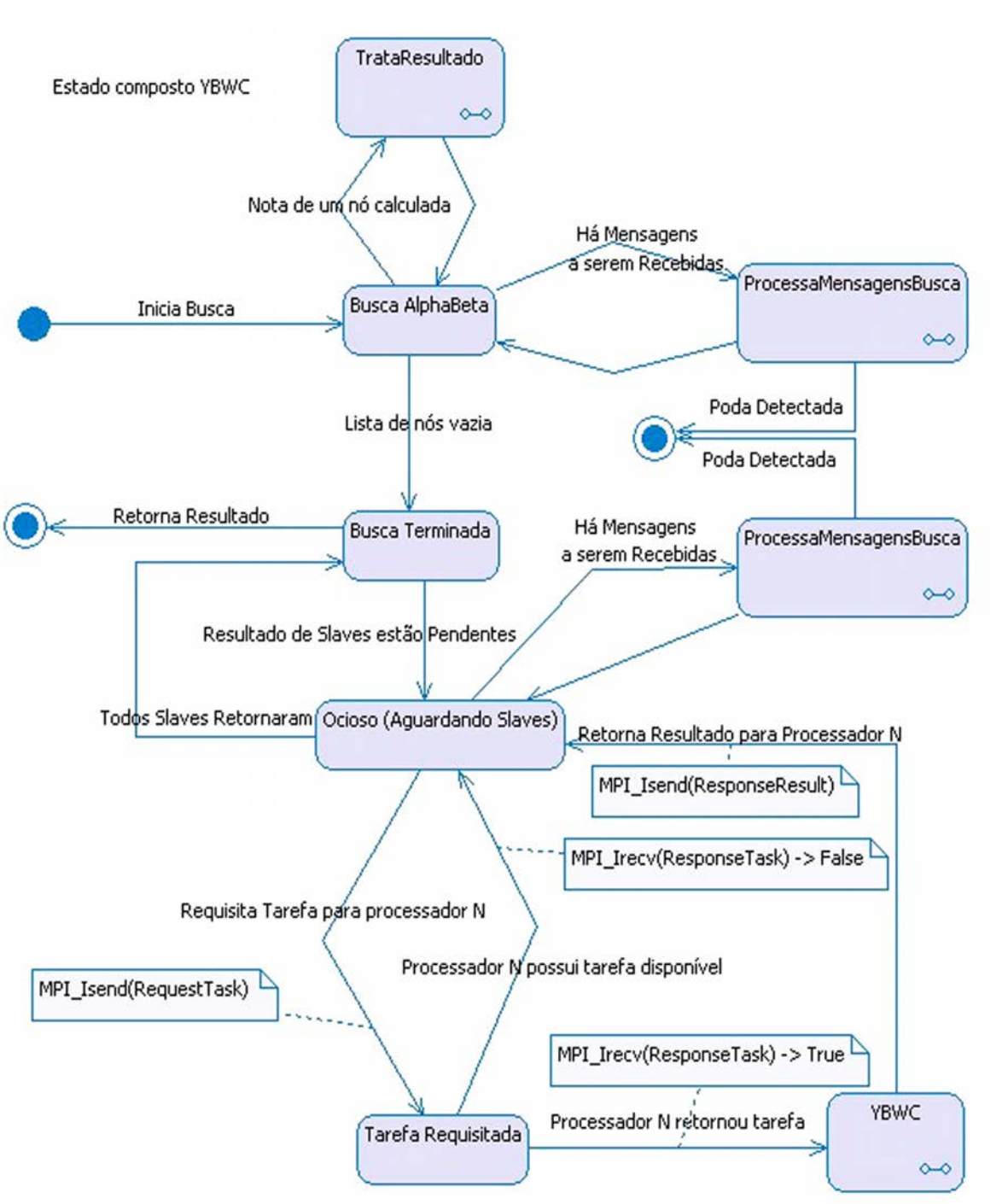

FIGURA 6.12: Diagrama de estados YBWC

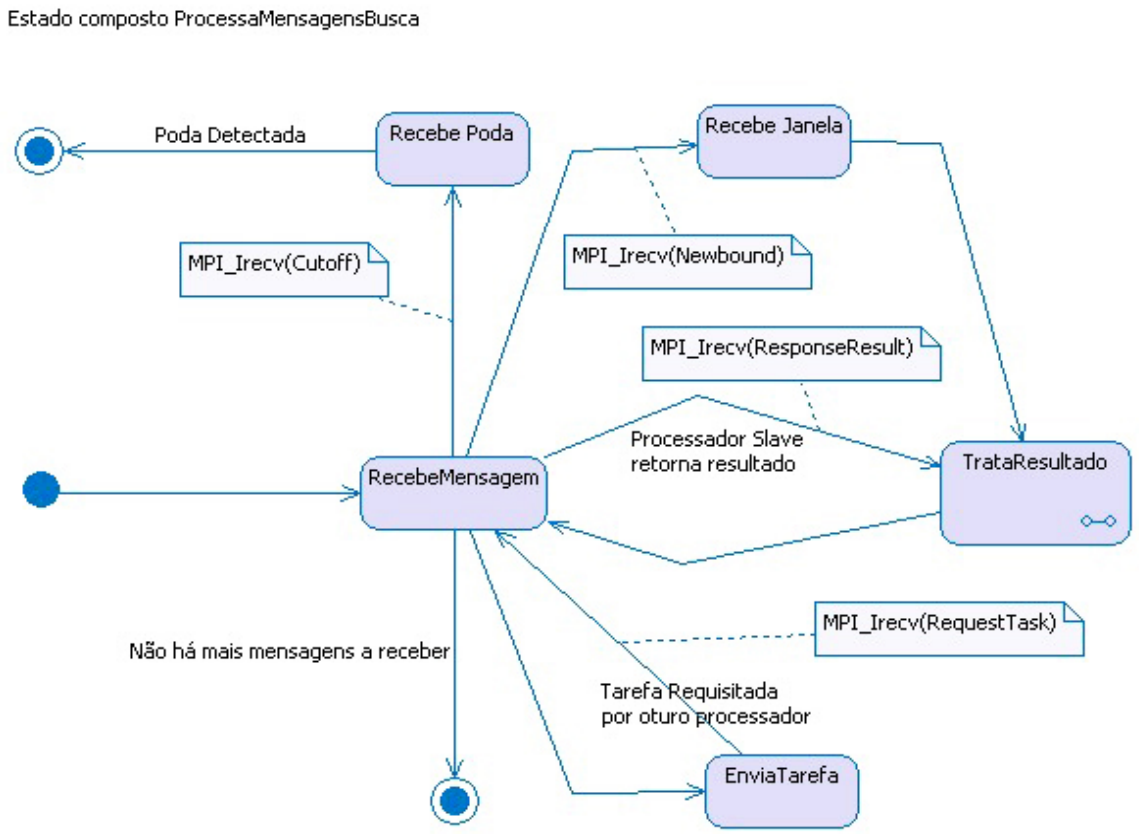

FIGURA 6.13: Diagrama de estados ProcessaMensagensBusca

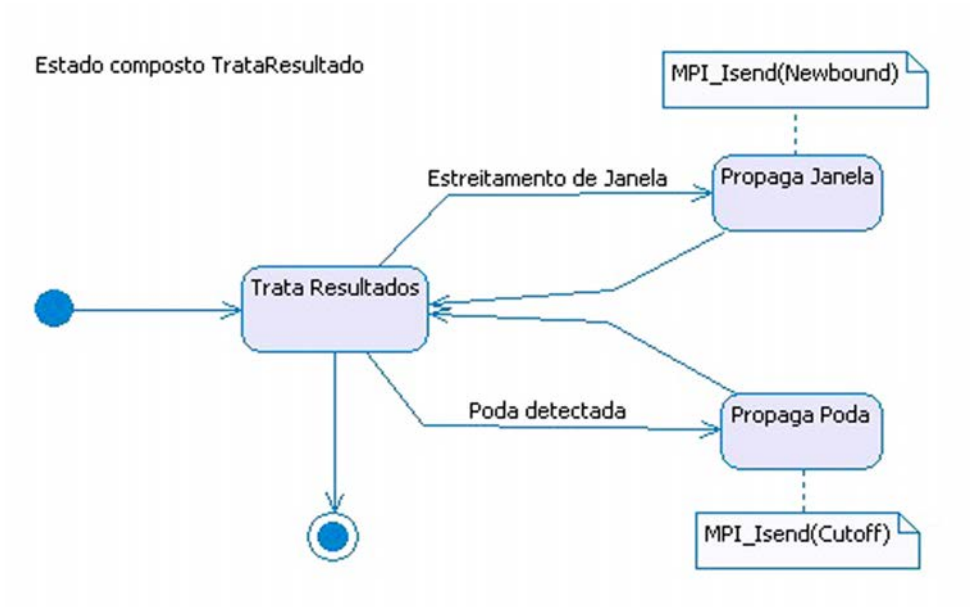

FIGURA 6.14: Diagrama de estados TrataResultado

6.3 Módulo de Aprendizagem

O Módulo de Aprendizagem do *D-MA-Draughts* consiste no ajuste de pesos das redes MLP´s jogadoras que avaliam os estados de tabuleiro gerados pelo algoritmo de busca YBWC. A atualização dos pesos é feita através dos métodos das Diferenças Temporais TD(λ). Ressalta-se que a dinâmica de treinamento, tanto do IIGA, quanto dos agentes de final de jogo é a mesma diferindo apenas em como os pesos da rede são inicializados e nos estados de tabuleiro que cada agente utiliza para realizar o processo de aprendizagem conforme será detalhado na seção 6.3.4.

A performance do processo de aprendizado depende diretamente de uma representação apropriada do estado do tabuleiro do jogo. Conforme visto na arquitetura do ciclo de treinamento do *D-MA-Draughts* (veja seção 6.1), os tabuleiros avaliados pelas redes MLP´s são representados por características (*features*) do próprio jogo de damas. O módulo responsável por esta conversão é o NET-FEATUREMAP, que recebe como entrada a representação vetorial do estado corrente do tabuleiro do jogo (veja seção 2.10.1) e o converte para a representação por características. Estas características são baseadas no trabalho de Samuel [6], [7], que propôs 28 características para representação dos estados do jogo de damas, conforme detalhado na seção 2.10.2. Em resumo, cada característica tem um valor absoluto que representa sua medida analítica em um determinado estado do tabuleiro. Tal valor é convertido em bits significativos, ou seja, conversão numérica entre as bases decimal e binária, onde em conjunto com os demais bits das outras características presentes na conversão constituem a sequência binária de saída do módulo NET-FEATUREMAP a ser representada na entrada da rede neural. Desta forma, o número de neurônios que irão compor a entrada da rede MLP é definido pela soma de todos os bits representativos das características que formam o mapeamento NET-FEATUREMAP.

Neste contexto, quanto maior o número de características utilizadas para representação do estado do tabuleiro do jogo, mais precisa é essa representação, fato que permite ao agente uma melhor percepção do ambiente que está atuando e, consequentemente, propicia uma tomada de decisão mais acertada. A fim de atuar em tal situação, o *D-MA-Draughts* investigou os ganhos obtidos ao representar seu mapeamento NET-FEATUREMAP com um número maior de características, mais precisamente, uma variação entre 14 e 16, uma vez que seu predecessor, *D-VisionDraughts*, fez uso de 12 características para representar os estados do tabuleiro (características **F1** a **F12** da tabela 6.2). Esta alteração foi efetuada para melhorar a percepção do agente em relação ao ambiente. O conjunto das características utilizadas no *D-MA-Draughts* é listado na tabela 6.2. As novas características incluídas foram: *diagonalMoment*, *threat*, *take* e *dyke*. Este aumento do número de características, inevitavelmente, influenciou no desempenho do sistema no que diz respeito ao aumento da carga de processamento, todavia, o aumento da capacidade de processamento do sistema fez com que este fato não fosse um fator que limitasse o *D-MA-Draughts* (veja a seção 6.2).

A próxima seção explica como é realizado o cálculo da predição (avaliação) do estado de tabuleiro apresentado à MLP para a escolha da melhor ação. Na sequência, a seção 6.3.2, será apresentado como é realizado o reajuste de pesos na MLP; a seção 6.3.3 descreve o processo de treinamento por

CARACTERÍSTICAS	N° BITS
F1: *PieceAdvantage*	4
F2: *PieceDisadvantage*	4
F3: *PieceThreat*	3
F4: *PieceTake*	3
F5: *Advancement*	3
F6: *DoubleDiagonal*	4
F7: *BackRowBridge*	1
F8: *CentreControl*	3
F9: *XCentreControl*	3
F10: *TotalMobility - MOB*	4
F11: *Exposure*	3
F12: *KingCentreControl*	3
F13: *DiagonalMoment*	3
F14: *Threat*	3
F15: *Taken*	3
F16: *Dyke*	1

Tabela 6.2: Conjunto de Características implementadas no jogador *D-MA-Draughts*

self-play com clonagem; e a seção 6.3.4 apresenta as particularidades do treinamento do agente IIGA e dos agentes de final de jogo.

6.3.1 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 6.15. 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 módulo de mapeamento NET-FEATUREMAP mostrado na figura 6.1;

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 $1 \leq \mathrm{l} \leq 2$, da seguinte forma:

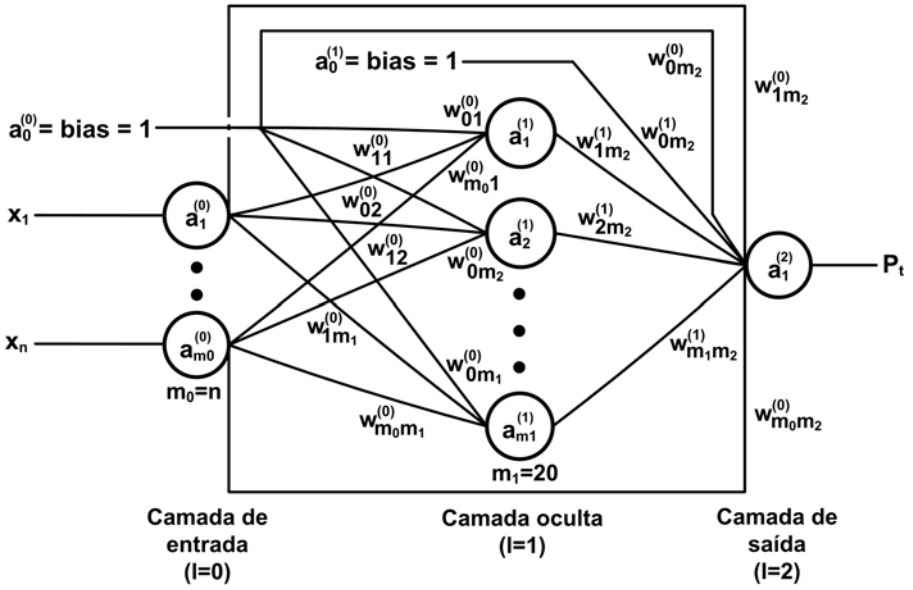

FIGURA 6.15: Rede Neural utilizada pelo *D-MA-Draughts*

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

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 tangente hiperbólica definida por $g_j(x) = \frac{2}{(1+e^{-2x})} - 1$;

4. a predição P_t retornada pela rede neural para o estado S_t é $P_t = a_1^{(2)} = g_1(in_1^{(2)})$.

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 do agente (representada pelo retorno do valor +1 pelo ambiente), derrota do agente (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).

6.3.2 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) a cada vez que o agente executa uma ação (um movimento). Considerando um conjunto de movimentos (análogo a um conjunto de ações efetuadas por um agente sobre o ambiente, veja seção 2.2.1) que o agente executa durante todo o jogo $\{M_0, ..., M_{i-1}, M_i, M_{i+1}, ..., M_t\}$, onde M_t se refere a um estado final de um episódio (aqui um estado final do jogo), as seguintes ações podem ocorrer: após a execução de M_t ou a rede neural é recompensada por uma boa performance (isto é, recebe uma recompensa positiva (+1) do ambiente que é associada a estados de vitória) ou é punida por uma má performance (isto é, recebe do ambiente uma punição (-1) do ambiente que é associada a um estado de derrota).

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 YBWC 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 [39]:

$$\begin{aligned} w_{ij}^{(l)} &= w_{ij}^{(l)}(t-1) + \Delta w_{ij}^{(l)}(t) \\ &= w_{ij}^{(l)}(t-1) + \alpha^{(l)}.(P_{t+1} - P_t).\sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k \\ &= w_{ij}^{(l)}(t-1) + \alpha^{(l)}.(P_{t+1} - P_t).elig_{ij}^{(l)}(t), \end{aligned} \tag{6.6}$$

onde $\alpha^{(l)}$ é o parâmetro da taxa de aprendizagem na camada.

Caixeta e Julia [18] utilizaram 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 $elig_{ij}^{(l)}(t)$ é único para cada peso sináptico $w_{ij}^{(l)}(t)$ da rede neural e representa o traço de eligibilidade das predições

calculadas pela rede para os estados resultantes de ações executadas pelo agente desde o instante temporal 1 do jogo até o instante temporal t; $\nabla_w P_k$ representa a derivada parcial de P_k em relação aos pesos da rede no instante k. Cada predição P_k é uma função dependente do vetor de entrada $\overrightarrow{X(k)}$ e do vetor de pesos $\overrightarrow{W(k)}$ da rede neural no instante temporal k. O termo 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 [17], [15], [77] 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 ($l = 2$):

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

     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) [28].
   - para os pesos associados às ligações entre as camadas de entrada ($l = 0$) e oculta ($l = 1$):

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

     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 ($l = 2$):

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

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:

$$\Delta w_{ij}^{(l)}(t) = \alpha^{(l)}.e(t).elig_{ij}^{(l)}(t), \tag{6.7}$$

onde o parâmetro de aprendizagem $\alpha^{(l)}$ é definido por Caixeta [18] como:

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

6. existe um problema típico associado ao uso de redes neurais no qual a convergência não é garantida para o melhor valor [15]. Caixeta [18], assim como Lynch [28], [16] utilizou o termo momento μ para tentar solucionar esse tipo de problema. Para isso, ele empregou uma checagem de direção na equação 6.6 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:

$$w_{ij}^{(l)}(t) = w_{ij}^{(l)}(t-1) + \Delta w_{ij}^{(l)}(t); \tag{6.8}$$

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

- calcule $\Delta w_{ij}^{(l)}(t)$ pela equação 6.6;
- se ($\Delta w_{ij}^{(l)}(t) > 0$ e $\Delta w_{ij}^{(l)}(t-1) > 0$) ou $\Delta w_{ij}^{(l)}(t) < 0$ e $\Delta w_{ij}^{(l)}(t-1) < 0$), então faça:

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

6.3.3 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 *D-MA-Draughts* 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 *D-MA-Draughts* 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.

6.3.4 Particularidades no treinamento dos agentes do D-MA-Draughts

As técnicas utilizadas no processo de aprendizado são as mesmas para todos os agentes do *D-MA-Draughts*, todavia, há algumas particularidades no treinamento do IIGA e dos agentes de final de jogo devido ao foco de cada um em uma partida. Estas particularidades são:

1. Como são inicializados os pesos da rede MLP;

2. Quais os estados de tabuleiro são utilizados no processo de treinamento.

No caso do agente IIGA, seu treinamento é conduzido a partir de um tabuleiro em estado inicial padrão de um jogo de damas e os pesos da rede são iniciados aleatoriamente. Como dito na seção anterior, o treinamento por *self-play* com clonagem é realizado por uma sequencia de jogos contra uma cópia do próprio jogador durante um determinado número de seções de forma a atualizar os pesos da rede neural MLP. Particularmente, o IIGA fez uso de 10 seções de 10 jogos.

Como os agentes de final de jogo tem o objetivo de atuar em fases finais da partida, mais precisamente, em estados de tabuleiro com 12 peças ou menos, não é conveniente adotar o mesmo mecanismo de inicialização dos pesos da rede neural utilizado no IIGA. Além disso, o treinamento não é realizado a partir de um único estado de tabuleiro e sim a partir dos estados de tabuleiro

clusterizados no processo descrito na seção 5.2. Todos os estados contidos nos *clusters* contém 12 peças. Por estes motivos, os pesos iniciais das redes MLP´s dos agentes de final de jogo correspondem aos pesos do IIGA já treinado. A justificativa para este procedimento é que estes agentes não atuarão desde o início padrão de um jogo de Damas, logo a inicialização aleatória comprometeria a eficiência do jogador, visto que este se encontraria em desvantagem em relação a um adversário que se preparou para uma partida completa. Uma seção s de treinamento de um agente de final de jogo é composta por uma quantidade q de jogos e é realizada para cada um dos modelos de tabuleiro mt presentes no *cluster* i em questão. Cada *cluster* servirá de base para a criação de um agente de final de jogo (veja seção 5.2.1). Desta forma, pode-se definir a quantidade de jogos de um processo de treinamento referente a um *agente* i como s x q x mt. Particularmente, no processo de treino de cada um dos agentes de final de jogo do *D-MA-Draughs* foram realizadas 3 seções compostas por 6 jogos para cada um dos 25 modelos de tabuleiros que compõem o *cluster* i, totalizando 450 jogos de treinamento para cada agente.

Capítulo 7

Resultados Experimentais

Esta seção apresenta e analisa os resultados obtidos pelo *D-MA-Draughts* em relação a seus predecessores: *MP-Draughts* e *D-VisionDraughts*. Neste sentido, primeiramente, a seção 7.1 apresenta o ambiente de desenvolvimento sobre o qual o *D-MA-Draughts* foi construído. Para os fins avaliativos, foram criados diversos cenários, onde cada um objetiva apresentar os progressos obtidos durante o desenvolvimento do presente trabalho. Neste contexto, a seção 7.2 apresenta os cenários cuja meta é estimar as melhorias obtidas individualmente pelos agentes do *D-MA-Draughts* com as otimizações aqui propostas. Para tanto, tais cenários se concentram, particularmente, na análise do desempenho atingido pelo agente IIGA do *D-MA-Draughts*, que, por sua vez, corresponde a um refinamento do *D-VisionDraughts*. Convém relembrar que a arquitetura do IIGA é similar à dos demais agentes. A seção 7.3 apresenta os cenários que avaliam o desempenho global dos agentes do *D-MA-Draughts* em termos do tempo de treinamento dos agentes e da atuação do sistema em torneios.

7.1 Ambiente de Execução dos Testes

Esta seção descreve as configurações do ambiente sobre o qual o *D-MA-Draughts* foi implementado.

Em relação a hardware, o *D-MA-Draughts* contou com uma infra-estrutura (conforme ilustração da Figura 6.2) composta por quatro estações de igual configuração: processador Intel QuadCore e 4GB de memória RAM. Todas as estações foram interligadas por um *Switch Gigabit Ethernet*.

O *D-MA-Draughts* foi desenvolvido na linguagem de programação C++ e compilado com a versão 4.4.1 do GCC (*GNU Compiler Collection*). Além disso, foi utilizada a biblioteca OpenMPI - versão 1.4.3 - na inclusão da interface de troca de mensagens (MPI) no sistema.

7.2 Melhoria na Arquitetura Individual dos Agentes do D-MA-Draughts

Como descrito no capítulo 4, cada agente do *D-MA-Draughts* é treinado segundo a arquitetura base do *D-VisionDraughts*, todavia, duas melhorias foram adicionadas: o aumento do número de características (*features*) na composição do mapeamento NET-FEATUREMAP para representação dos estados de tabuleiro e o aumento na capacidade de processamento do sistema com a adição de novos processadores. Com o intuito de avaliar o ganho individual de cada agente com a inserção de tais otimizações, os 5 cenários abaixo se propõem a estimar, particularmente e a título de exemplo, o desempenho do agente IIGA do *D-MA-Draughts* comparado ao *VisionDraughts* (ver seção 3.2.2) e ao do *D-VisionDraughts*. Nos cenários de I a III, onde se pretende avaliar o desempenho de processamento, o número de características não foi alterado: manteve-se fixo em 12, tal como no *D-VisionDraughts* e no *VisionDraughts*. Já no cenário IV, executado para avaliar o impacto no treinamento trazido por alterações na representação NET-FEATUREMAP, o número de características variou entre 12 e 16. O cenário V teve como objetivo avaliar o desempenho geral do IIGA em jogos normais de disputa (após treinado), operando com 14 e com 16 características.

Cenário I Este cenário verifica o ganho obtido com o gradual aumento do número de processadores no IIGA. Para isso, foram realizados treinamentos limitados por um período de quatro horas, onde cada um contou com um número de processadores diferente. As configurações aplicadas a cada treinamento foram as seguintes:

- Estado de tabuleiro inicial padrão em jogos de Damas;
- Busca em profundidade primeira (limitada à profundidade 12);

Pode-se observar que, neste cenário, o número de características não foi incrementado, visto que o objetivo foi verificar apenas o ganho obtido em relação à capacidade de processamento. Desta forma, foi mantido o número de características utilizadas no predecessor do IIGA (*D-VisionDraughts*). A Tabela 7.1 apresenta os resultados obtidos pela aplicação deste cenário onde o agente contou com 10, 14 e 16 processadores. A seção onde o agente atuou com 10 processadores corresponde ao *D-VisionDraughts*. É possível notar que, nesta seção, em virtude das limitações de tempo e busca, foi possível executar apenas 48 jogos de treinamento. As seções seguintes, isto é, aquelas em que o agente atuou com 14 e 16 processadores, o sistema demonstrou um ganho significativo, uma vez que 64 e 72 jogos de treinamento foram executados, respectivamente. Desta forma, este primeiro cenário mostrou que a performance do IIGA, em termos de quantidade de jogos de treinamento executados por um período de quatro horas (comparado com o desempenho do *D-VisionDraughts*, ou seja, operando apenas com 10 processadores), melhorou em 33% quando operando com 14 processadores e em 50% quando operando com 16 processadores.

Como o melhor desempenho ocorreu com 16 processadores, adotou-se esta quantidade de processadores para compor os demais testes realizados para o sistema apresentado neste trabalho e, de fato, tornou-se a quantidade de processadores oficial adotada pelo IIGA e demais agentes do *D-MA-Draughts*.

TABELA 7.1: Número de Processadores X Número de Jogos

Número de Processadores	Número de Jogos
10	48
14	64
16	72

Cenário II O segundo cenário foi definido a fim de avaliar o ganho obtido em termos de *look-ahead* obtido pelo IIGA. Neste cenário, as condições aplicadas ao cenário I foram mantidas, exceto no que diz respeito à estratégia de busca, uma vez que agora foi utilizado profundidade iterativa, ao invés da profundidade de busca limitada. Neste cenário foi possível constatar que o IIGA conseguiu explorar até dois níveis mais profundos na árvore de jogo em relação ao *D-VisionDraughts*.

Cenário III O terceiro cenário avaliou o tempo de treinamento obtido no IIGA em relação aos seus predecessores: *VisionDraughts* e *D-VisionDraughts*. Tal cenário foi composto por 10 seções de treinamento, cada uma composta por 10 jogos, conforme a definição da estratégia de treinamento apresentada na seção 6.3. A profundidade de busca foi limitada a 12. A figura 7.1 mostra a eficiência atingida pelo IIGA devido à adição dos novos processadores. Note que o tempo de treinamento requerido pelo *VisionDraughts* é aproximadamente 50% maior em relação ao requerido pelo *D-VisionDraughts*. Por outro lado, o IIGA se mostrou 70% e 43% mais rápido que o *VisionDraughts* e o *D-VisionDraughts*, respectivamente.

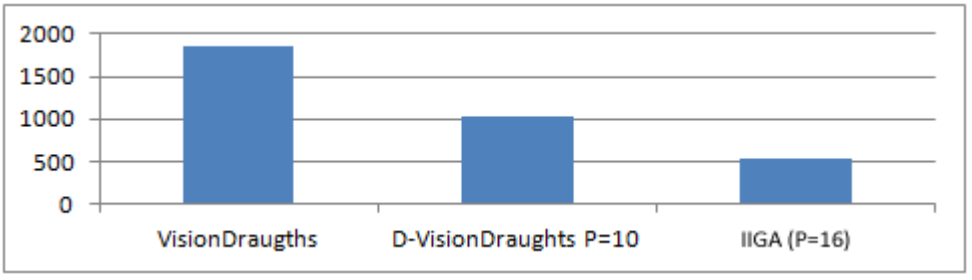

FIGURA 7.1: Tempo de treinamento (em minutos) dos sistemas: VisionDraughts, D-VisionDraughts (10 processadores) e IIGA do D-MA-Draughts (16 processadores)

Cenário IV O quarto cenário checou a performance do IIGA durante o treinamento, com a inserção gradual de características no mapeamento NET-FEATUREMAP. Para tanto, foram realizados testes com 12, 14 e 16 características. A configuração deste cenário, assim como no cenário III, também foi composto por 10 seções de 10 jogos e a profundidade de busca foi limitada a 12.

A figura 7.2 mostra que o tempo de treinamento do jogador aumenta proporcionalmente ao número de características incluídas no mapeamento NET-FEATUREMAP. De fato, o tempo de treinamento onde o IIGA contou com 12 características (534 minutos) foi 33.7% menor que o tempo de treinamento onde o IIGA utilizou 14 características e foi menor 49.8% que o tempo de treinamento com 16 características.

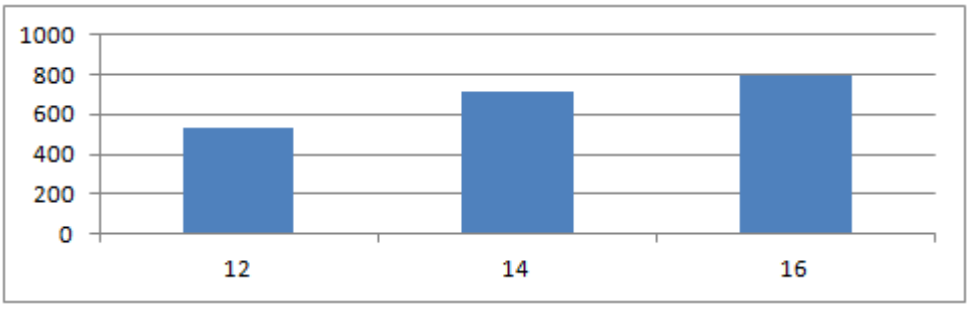

FIGURA 7.2: Tempo de treinamento (em minutos) X Número de características (*features*)

Cenário V Este cenário executa um torneio onde o IIGA joga contra o *D-VisionDraughts* e o *VisionDraughts*. O objetivo foi verificar se as melhorias implementadas no IIGA (adição do número de processadores e características) de fato contribuíram para a melhoria da performance do jogador em termos de número de vitórias.

Cada torneio foi composto por 40 jogos. O primeiro torneio foi executado entre o IIGA e o *VisionDraughts*. Como mostrado na tabela 7.2, o IIGA composto por 14 características em seu mapeamento NET-FEATUREMAP, não obteve nenhuma derrota e obteve 6 vitórias (o que corresponde a uma taxa superior de 15% em vitórias). Por outro lado, o IIGA composto por 16 características obteve 7 vitórias, o que corresponde a uma taxa superior de 17.5% de vitórias em relação ao seu oponente.

O segundo torneio, cujo os resultados são mostrados na tabela 7.3 é disputado entre o IIGA e o *D-VisionDraughts* (composto por 10 processadores). Os resultados mostram que a taxa de vitória do IIGA em ambos as situações, ou seja, quando ele é composto por 14 ou 16 características, é 10% superior ao *D-VisionDraughts*.

TABELA 7.2: Resultados do IIGA do D-MA-Draughts (P=16) contra o VisionDraughts

Número de características	**Vitórias**	**Empates**	**Derrotas**
14	6	34	0
16	7	32	1

Observe que não houve um ganho expressivo em relação a vitórias quando comparado o IIGA com 14 e 16 características. Por outro lado, o tempo de treinamento exigido quando utilizado 14 características é menor do que quando são utilizadas 16 características. Portanto, neste momento, foram definidas as configurações do IIGA: atuará em um ambiente composto por 16 processadores

Tabela 7.3: Resultados do IIGA do D-MA-Draughts (P=16) contra o D-VisionDraughts(P=10)

Número de características	Vitórias	Empates	Derrotas
14	4	35	1
16	4	36	0

e seu mapeamento NET-FEATUREMAP será composto por 14 características. Ressalta-se que esta também será a configuração adotada para cada um dos agentes de final de jogo.

7.3 Verificando a performance do D-MA-Draughts

Esta seção apresenta a avaliação da performance do *D-MA-Draughts* comparado com seus predecessores que lhe serviram de inspiração: *MP-Draughts* e *D-VisionDraughts*. Para fins de simplificação, a tabela 7.4 resume os sistemas envolvidos nos cenários de VI a IX apresentados a seguir.

Tabela 7.4: Resumo dos sistemas MP-Draughts, D-VisionDraughts e D-MA-Draughts

Agente	Distribuído	Multiagente	N. de Características
MP-Draughts	Não	Sim	14
D-VisionDraughts	Sim	Não	12
D-MA-Draughts	Sim	Sim	14

Cenário VI Neste cenário foi avaliado o impacto da distribuição de busca sobre o sistema multiagente. Para isso, foi avaliado o tempo de treinamento total dos sistemas *MP-Draughts* e *D-MA-Draughts*, visto que ambos possuem o mesmo número de agentes.

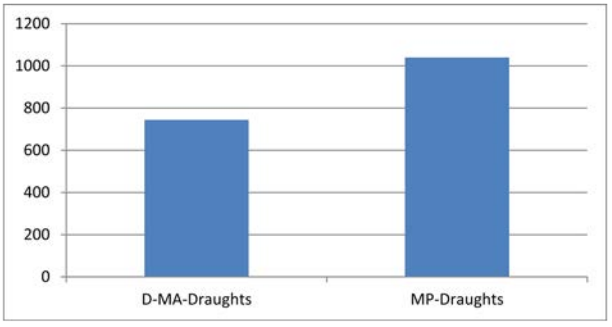

Figura 7.3: Tempo de treinamento (horas) do D-MA-Draughts e MP-Draughts

Para a realização deste teste, cada um dos agentes adotou, em seu mecanismo de busca, profundidade fixa igual a 12. Ambos os sistemas utilizaram o mesmo conjunto de *clusters* de estados de tabuleiros para treinamento de seus agentes de final de jogo.

A figura 7.3 ilustra um gráfico que demonstra que o tempo de treinamento dos agentes do *D-MA-Draughts* (744 horas) foi 28.46% menor do que o tempo exigido pelo *MP-Draughts*. Tal fato se deve à paralelização do mecanismo de busca, uma vez que é possível se chegar no mesmo nível da árvore de busca do jogo em menor espaço de tempo.

Cenário VII Outro ganho que a distribuição da busca proporcionou ao *D-MA-Draughts* foi uma visão mais profunda da árvore do jogo (*look-ahead*). A fim de comprovar tal fato foi realizado um novo treinamento para os agentes tanto do *D-MA-Draughts* quanto do *MP-Draughts*. No entanto, desta vez foi adotado no mecanismo de busca o aprofundamento iterativo. Neste caso, cada jogada foi limitada a 10 segundos. Como no decorrer do jogo não há garantia que o agente vá atingir sempre a mesma profundidade, visto que sua busca dependerá do estado corrente da partida, foi feita uma média dos níveis mais profundos que cada agente conseguiu atingir dentro do mesmo espaço de tempo.

Neste contexto, foi possível constatar que o *D-MA-Draughts* conseguiu em determinados momentos do jogo atingir uma profundidade até 4 vezes maior que o *MP-Draughts*.

Cenário VIII A fim de verificar a performance do *D-MA-Draughts* em jogos normais de disputa, ou seja, em partidas reais, foram realizados torneios contra o *MP-Draughts* e o *D-VisionDraughts* em cada uma de suas dinâmicas de atuação de jogo conforme apresentadas na seção 5.3.2. Cada torneio foi composto por 40 partidas. Foram feitas diversas combinações no que diz respeito à estratégia de busca no treinamento dos agentes (treinados em profundidade de busca fixa ou iterativa), bem como na estratégia de busca nos jogos normais (em profundidade fixa ou iterativa).

Neste ponto, o leitor poderia imaginar que não há sentido um torneio entre o *D-MA-Draughts* e o *MP-Draughts* quando ambos foram treinados utilizando a mesma profundidade de busca, uma vez que isso produziria uma situação em que os agentes de ambos os sistemas seriam iguais (redes MLP´s com os mesmos pesos). Todavia, isso não ocorre, pois, os IIGA´s de ambos são distintos. De fato, no *MP-Draughts*, o IIGA é o próprio *VisionDraughts* (veja a seção 3.2.2), ao passo que o do *D-MA-Draughts* conta com 14 características, conforme estabelecido em função dos resultados obtidos nos torneios do cenário V. Consequentemente, como as redes de final de jogo são treinadas a partir da matriz de pesos do IIGA, os agentes de final de jogo de ambos os sistemas são distintos.

A tabela 7.5 apresenta os resultados dos torneios cujos jogos foram executados com o mecanismo de busca configurado com profundidade fixa igual a 10, onde o *D-MA-Draughts* joga usando a dinâmica I de interação entre os agentes. Ressalta-se que os torneios foram agrupados segundo o modo no qual os agentes foram treinados, ou seja, com profundidade fixa ou iterativa. É importante ressaltar que também foram executados torneios com profundidades de busca diferentes de 10 e que, neles, foram obtidos resultados bem próximos aos obtidos com limite 10.

Deve-se notar que nos torneios executados contra o *MP-Draughts*, o *D-MA-Draughts* obteve 17.5% de vitórias contra 12.5%, considerando-se treinamento em profundidade fixa, o que corresponde a uma taxa de 5% de superioridade em relação ao seu oponente. Para treinamento em profundidade

iterativa, o *D-MA-Draughts* obteve 20% de vitórias contra 7,5% de seu oponente. Neste caso, o *D-MA-Draughts* conseguiu superar o *MP-Draughts* em 12.5%, vantagem obtida em função do *look-ahead* mais profundo.

Em relação aos torneios executados contra o *D-VisionDraughts*, note que em ambos os casos de treinamento o *D-MA-Draughts* teve melhor performance. No primeiro caso de treinamento (profundidade de busca fixa), o *D-MA-Draughts* obteve 17.5% de vitória e, no segundo caso, 20%. No primeiro caso, a vantagem do *D-MA-Draughts* se deve ao incremento do número de características no mapeamento NET-FEATUREMAP e à sua arquitetura multiagente, uma vez que na dinâmica I, quando for atingido um estado de tabuleiro de final de jogo, um agente (escolhido pela rede Kohonen-SOM dentre os agentes de final de jogo) assumirá o papel de EGA e conduzirá a partida até o final. No segundo caso de treinamento, a vantagem do *D-MA-Draughts* é justificada, não apenas por sua arquitetura multiagente, mas também pelo número de processadores superior com o qual foi treinado (16 processadores contra 10 processadores utilizados no *D-VisionDraughts*).

Tabela 7.5: Dinâmica I do D-MA-Draughts X MP-Draughts e D-VisionDraughts jogando em profundidade fixa igual a 10

Método de treinamento	**Oponente**	**Vitórias**	**Derrotas**	**Empates**
Profundidade Fixa	MP-Draughts	7	5	28
	D-VisionDraughts	7	4	29
Profundidade Iterativa	MP-Draughts	8	3	29
	D-VisionDraughts	8	5	27

A tabela 7.6 também apresenta resultados de torneios em que o *D-MA-Draughts* joga de acordo com a dinâmica I. Todavia, diferentemente da tabela anterior, os jogos destes torneios foram executados com profundidade de busca iterativa. Cada jogada foi limitada a 10 segundos a fim de não prolongar a partida demasiadamente e nem prejudicar os agentes com jogadas cujo *look-ahead* não fosse muito profundo. A superioridade do *D-MA-Draughts* neste contexto é decorrente das otimizações inseridas na representação dos estados do tabuleiro e na capacidade de processamento. Observe que, contra o *MP-Draughts*, o *D-MA-Draughts* obteve 27,5% de vitórias (uma superioridade de 17.5%), considerando-se treinamento em profundidade fixa. Além disso, considerando-se treinamento em profundidade iterativa, obteve 30% de vitórias resultando em uma superioridade de 15%. Nos torneios contra o *D-VisionDraughts*, foram obtidas 30% de vitórias quando os agentes foram treinados em profundidade de busca fixa e 25% de vitórias quando os agentes foram treinados em profundidade de busca iterativa. Em ambos os casos, a superioridade do *D-MA-Draughts* foi de 7,5 em relação ao *D-VisionDraughts*.

Os resultados acima confirmam os bons resultados introduzidos com a associação da dinâmica I com as melhorias na representação de estados e na capacidade de treinamento.

As tabelas 7.7 e 7.8 apresentam os mesmos casos dos torneios apresentados nas tabelas 7.5 e 7.6, respectivamente. Todavia, nelas o *D-MA-Draughts* opera nos moldes da sua segunda dinâmica de atuação (dinâmica II).

Tabela 7.6: Dinâmica I do D-MA-Draughts X MP-Draughts e D-VisionDraughts jogando em profundidade de busca iterativa limitando a 10 segundos por jogada

Método de treinamento	**Oponente**	**Vitórias**	**Derrotas**	**Empates**
Profundidade Fixa	MP-Draughts	11	4	25
	D-VisionDraughts	12	9	19
Profundidade Iterativa	MP-Draughts	12	6	22
	D-VisionDraughts	10	7	23

Tabela 7.7: Dinâmica II do D-MA-Draughts X MP-Draughts e D-VisionDraughts jogando em profundidade fixa igual a 10

Método de treinamento	**Oponente**	**Vitórias**	**Derrotas**	**Empates**
Profundidade Fixa	MP-Draughts	11	7	22
	D-VisionDraughts	12	8	20
Profundidade Iterativa	MP-Draughts	17	11	12
	D-VisionDraughts	10	4	26

Tabela 7.8: Dinâmica II do D-MA-Draughts X MP-Draughts e D-VisionDraughts jogando em profundidade de busca iterativa limitando a 10 segundos por jogada

Método de treinamento	**Oponente**	**Vitórias**	**Derrotas**	**Empates**
Profundidade Fixa	MP-Draughts	12	7	21
	D-VisionDraughts	10	5	25
Profundidade Iterativa	MP-Draughts	13	8	19
	D-VisionDraughts	11	8	21

A tabela 7.7 apresenta os torneios executados com profundidade de busca fixa entre o *D-MA-Draughts* contra o *MP-Draughts* e *D-VisionDraughts*. Observe que contra o *MP-Draughts*, o *D-MA-Draughts* conseguiu 27,5% de vitórias, para treinamento com profundidade de busca limitada (uma superioridade de 10%). No torneio em que é considerada a profundidade de busca iterativa, o *D-MA-Draughs* atingiu 42,5% de vitórias (uma superioridade de 15%). É importante destacar a melhora da dinâmica II do *D-MA-Draughts* (em relação ao número de vitórias) que se mostrou 10% e 22,5% superior, respectivamente, nos casos de treinamento em profundidade fixa e iterativa (jogando contra o *MP-Draughts*).

Em relação ao *D-VisionDraughts*, o *D-MA-Draughts* atingiu 30% e 25% de vitórias segundo os dois casos apresentados na tabela 7.7, respectivamente. Note que nestes torneios também ocorreu uma melhora em relação à dinâmica I. A dinâmica II, em relação ao número de vitórias, mostrou-se 12,5% e 5% superior, respectivamente, nos casos de treinamento em profundidade fixa e iterativa (jogando contra o *D-VisionDraughts*).

A tabela 7.8 apresenta os resultados dos torneios executados segundo a dinâmica de jogo II e em profundidade de busca iterativa (limitada a 10 segundos por jogada). Em relação ao *MP-Draughts*, o *D-MA-Draughts* atingiu uma superioridade de 12,5% considerando ambos os casos de treinamento dos agentes. Quando comparado à dinâmica I, a melhora ficou em torno de 2,5% em ambos os casos.

Em contrapartida, nos torneios contra o *D-VisionDraughts*, considerando o treinamento em profundidade fixa, o desempenho obtido com a dinâmica II mostrou-se 5% inferior ao da dinâmica I, o que aponta para a necessidade de um melhor refinamento da etapa de treino. Por outro lado, considerando treinamento por aprofundamento iterativo, a dinâmica II voltou a superar em 2,5% a dinâmica I.

Cenário IX Este é o último cenário de teste tratado neste capítulo. O objetivo foi realizar uma análise sobre o problema de *loops* de final de jogo (veja seção 4). A tabela 7.9 mostra a quantidade de *loops* obtidos em torneios de 40 partidas disputados em profundidade de busca fixa. Para isso, tais torneios consideraram agentes treinados ora em profundidade fixa de 10, ora em profundidade iterativa limitada a 5 segundos. Foram envolvidos os jogadores *D-VisionDraughts*, *MP-Draughts* e *D-MA-Draughts*, este último operando em suas duas dinâmicas (DI e DII).

Tabela 7.9: D-VisionDraughts x MP-Draughts, D-MA-Draughts - DI (dinâmica de jogo I) e D-MA-Draughts - DII (dinâmica de jogo II)

Método de treinamento	**Oponente**	**Loops de Final de Jogo**
Profundidade Fixa	MP-Draughts	28
	D-MA-Draughts - DI	28
	D-MA-Draughts - DII	22
Profundidade Iterativa	MP-Draughts	14
	D-MA-Draughts - DI	12
	D-MA-Draughts - DII	8

Observe que, para treinamento em profundidade fixa, a DI do *D-MA-Draughts* não alterou a incidência dos *loops*. Já a DII conseguiu reduzi-la de 15%. Por outro lado, em treinamento em profundidade iterativa, a DI e a DII propiciaram uma redução de 15% e de 43% na ocorrência de *loops* de final de jogo.

Capítulo 8

Conclusão e trabalhos futuros

Este trabalho apresentou o *D-MA-Draughts*: um sistema multiagente que aprende por reforço e que atua em um ambiente de alto desempenho. O *D-MA-Draughts* integrou e refinou as arquiteturas de suas versões preliminares, *D-VisionDraughts* e *MP-Draughts*, do seguinte modo: adaptou o ambiente de alto desempenho do *D-VisionDraughts* à plataforma multiagente do *MP-Draughts*. Além disso, aumentou a acuidade da representação dos estados com a inserção de novas características no mapeamento NET-FEATUREMAP, bem como incrementou significativamente a capacidade de processamento com a inserção de novos processadores. Finalmente, introduziu como alternativa uma segunda dinâmica de atuação conjunta dos agentes que se mostrou bastante satisfatória na melhoria da performance do sistema em jogos de disputa. O aumento da capacidade de processamento mostrou-se eficiente para compensar a sobrecarga de trabalho ocasionada pela melhoria na representação dos estados. Além disso, esses processadores adicionais também permitiram uma melhor visão futura do jogo no momento da busca. Tais ganhos foram observados, tanto nos jogos-treino, quanto nos normais. A segunda dinâmica de atuação em jogos possibilitou uma maior cooperação dos agentes no andamento da partida, o que permitiu uma atuação mais eficaz nos torneios disputados. Todas essas melhorias foram comprovadas nos cenários de testes de validação efetuados. Quanto aos problemas de *loops* de final de jogo, foi constatado que, em determinados casos, a dinâmica DII de atuação em jogos do *D-MA-Draughts* os diminuiu significativamente.

No entanto, ao analisar os cenários de teste realizados, quanto os números obtidos nestes resultados são satisfatórios para constatar o desempenho deste agente jogador? Para responder a esta questão é importante relembrar que o *D-MA-Draughts* pertence a uma linha de pesquisa onde foram produzidos diversos jogadores automáticos. Inicialmente, verificava-se a evolução destes agentes jogadores comparando-os com o jogador *NeuroDraughts* (arquitetura base dos jogadores automáticos desta linha de pesquisa). Todavia, diversas técnicas de Inteligência Artificial foram empregadas nos agentes que antecederam o *D-MA-Draughts* (*LS-Draughts*, *VisionDraughts*, *MP-Draughts* e *D-VisionDraughts*) tornando-os cada vez mais eficientes e comprovadamente mais competitivos que o *NeuroDraughts*. Neste contexto, devido ao histórico de evolução destes agentes, obter números que apontem a superioridade do *D-MA-Draughts* em relação aos seus antecessores é satisfatório uma vez que atestam a evolução e qualidade do agente desenvolvido.

Deste modo, conclui-se que o *D-MA-Draughts* mantém a evolução dos predecessores de sua linha de pesquisa, no que diz respeito a desempenho. Todavia, como ponto a melhorar pode-se citar a exploração da abordagem das dinâmicas de atuação em partidas, de modo a considerar, por exemplo, mais fases de jogo e maior comunicação e colaboração entre os agentes.

8.1 Trabalhos Futuros

Apesar do bom desempenho do *D-MA-Draugths*, muitas técnicas ainda podem ser aplicadas em sua arquitetura na intenção de obter melhores resultados. Algumas sugestões de técnicas a serem aplicadas estão relacionadas a seguir:

- integrar ao *D-MA-Draughts* o módulo de algoritmo genético desenvolvido por Neto e Julia [17], [15], [77] no jogador *LS-Draughts* a fim de automatizar o processo de seleção das características para representação do mapeamento NET-FEATUREMAP em cada agente em função de suas especialidades. Atualmente, a escolha é feita manualmente;
- desenvolver uma interface apropriada, como a "*Checker-Board*" [Fierz], que permita ao *D-MA-Draughts* jogar com outros jogadores automáticos como o Cake [Fierz] e até mesmo com humanos em jogos "on line";
- substituir a utilização de redes Kohonen-SOM por redes adaptativas ART [78] no processo de agrupamento dos estados de tabuleiro de final de jogo, bem como na dinâmica de jogo envolvendo estes agentes de modo a automatizar a detecção do número de *clusters* apropriado.
- explorar a dinâmica de comunicação entre os agentes do jogo em diversos momentos da partida e não apenas em estados caracterizados como de final de jogo.

Referências Bibliográficas

[1] Gerhard Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, 1999. ISBN 978-0-262-23203-6.

[2] André Ricardo Gonçalves. Redes neurais artificiais, 2012. URL http://www.dca.fee.unicamp.br/~andreric/arquivos/pdfs/redes_neurais.pdf.

[3] D. Hubel and T. Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *Journal of Physiology*, 1962.

[4] J. Schaeffer, N. Burch, Y. Bjornsson, A. Kishimoto, M. Muller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 2007. ISSN 1144079+.

[5] Pedro Campos and Thibault Langlois. Abalearn: Efficient self-play learning of the game abalone. In *INESC-ID, Neural Networks and Signal Processing Group*, 2003.

[6] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, 3(3):210–229, jul 1959, Riverton, NJ, USA, IBM Corp. ISSN 0018-8646. URL http://dx.doi.org/10.1147/rd.33.0210.

[7] A. L. Samuel. Some studies in machine learning using the game of checkers. ii: recent progress. *IBM J. Res. Dev.*, 11(6):601–617, nov 1967, Riverton, NJ, USA, IBM Corp. ISSN 0018-8646. URL http://dx.doi.org/10.1147/rd.116.0601.

[8] J. V. Neumann and O. Morgenstern. *Theory of games and economic behavior.* teste, 1944.

[9] D. B. Fogel and K. Chellapilla. Verifying anaconda´s expert rating by competing agains chinook: experiments in co-evolving a neural checkers player. *Neurocomputing*, 42(1-4):69–86, 2002.

[10] H. Jaap van den Herik, Jos W. H. M. Uiterwijk, and Jack van Rijswijck. Games solved: now and in the future. *Artif. Intell.*, 134(1-2):277–311, jan 2002, Elsevier Science Publishers Ltd. ISSN 0004-3702.

[11] C. H. C. Ribeiro and S. T. Monteiro. Aprendizagem da navegação em robôs móveis a partir de mapas obtidos autonomamente. *IV Encontro Nacional de Inteligência Artificial (ENIA)*, 2003.

[12] L. Thorpe and C. W. Anderson. Traffic light control using sarsa with three state representations. *Technical Report, IBM Corporation*, 1996, Boulder, CO, USA.

[13] M. Wiering. Multi-agent reinforcement learning for traffic light control. *17th International Conf. on Machine Learning*, pp. 1151–1158, 2000, San Francisco, CA.

[14] M. A. Walker. An application of reinforcement learning to dialogue strategy in a spoken dialogue system for email. *Artificial Intelligence Research 12*, pp. 387–416, 2000.

[15] H. C. Neto. Ls-draughts - um sistema de aprendizagem de jogos de damas baseado em algoritmos genéticos, redes reurais e diferenças temporais. Master's thesis, Faculdade de Computação - Universidade Federal de Uberlândia, Uberlândia, Brasil, 2007.

[16] M. Lynch. An application of temporal difference learning to draughts. Master's thesis, University of Limerick, Ireland, 1997.

[17] H. C. Neto and R. M. S. Julia. Ls-draughts - a draughts learning system based on genetic algorithms, neural network and temporal differences. *IEEE Congress on Evolutionary Computation*, pp. 2523–2529, 2007.

[18] G. S. Caexeta. Vision-draughts - 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. Master's thesis, Faculdade de Computação - Universidade Federal de Uberlândia, Uberlândia, Brasil, 2008.

[19] G. S. Caixeta and R. M. S. Julia. A draughts learning system based on neural networks and temporal differences: The impact of an efficent tree-search algorithm. *SBIA 2008, LNAI Springer Verlag, New York*, 5249:73–82, 2008.

[20] V. A. R. Duarte. Mp-draughts: Um sistema multiagente de aprendizagem automática para damas baseado em redes neurais de kohonen e perceptron multicamadas. Master's thesis, Faculdade de computação - Universidade Federal de Uberlândia, Uberlândia, Brasil, 2009.

[21] V. A. R. Duarte and R. M. S. Julia. Mp-draughts: a multiagent reiforcement learning system based on mpl and kohonen-som neural networks. *IEEE Internation Conference on Systems, Man and Cybernetics*, pp. 2270–2275, 2009, SMC´09, IEEE Press.

[22] A. R. A. Barcelos, R. M. S. Julia, and R. M. Jr. D-visiondraughts: a draughts player neural network that learns by reinforcement a high performance environment. *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2011.

[23] A. R. A. Barcelos. D-visiondraughts: Uma rede neural jogadora de damas que aprende por reforço em um ambiente de computação distribuída. Master's thesis, Universidade Federal de Uberlândia, 2011.

[24] J. Schaeffer. Man versus machine: The silicon graphics world checkers championship. 1992.

[25] J. Schaeffer, R. Lake, P. Lu, and M. Bryant. Chinook: The world man-machine checkers champion. *AI Magazine 17*, pp. 21–29, 1996.

[26] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Printice Hall, 2nd edition edition, 1998.

[27] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press, 1998. ISBN 0262193981.

[28] M. Lynch and N. Griffith. Neurodraughts: the role of representation, search, training regime and architecture in a td draughts player. *Eighth Ireland Conference on Artificial Intelligence*, pp. 67–72, 1997, Ireland. URL http://iamlynch.com/nd.html.

[29] Lidia B. P Tomaz, Rita M. S. Julia, and Ayres R. A. Barcelos. Improving the accomplishment of a neural network based agent for draughts that operates in a distributed learning environment. *IEEE IRI 2013*, pp. 262–269, 2013.

[30] S. Russell and P. Norvig. *Inteligência Artificial - Uma abordagem Moderna*. Campus, 2ª edition, 2004.

[31] M. Wooldridge and N. R. Jennings. *Intelligent Agents: Theory and Practice.*, volume 10. 1995.

[32] Franco Zambonelli, R. N. Jennings, and M. Wooldridge. Organisational abstractions for the analysis and design of multi-agent systems. pp. 127–141. Springer-Verlag, 2000.

[33] Gerhard Weiss. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT Press, Cambridge, MA, USA, 1999.

[34] B. Moulin and Chaib-Draa Brahim. An overview of distributed artificial intelligence. *Foundations of distributed artificial intelligence*, 1996.

[35] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[36] W. Martins, U. R. Afonseca, L. E. G. Nalin, and V. M. Gomes. Tutoriais inteligentes baseados em aprendizado por reforço: Concepção, implementação e avaliação empírica. *Simpósio Brasileiro de Informática na Educação - SBIE - Mackenzie*, 2007.

[37] M. E. Harmon and S. S. Harmon. Reinforcement learning: A tutorial, 1996.

[38] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072.

[39] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Mach. Learn.*, 3(1):9–44, aug 1988, Hingham, MA, USA, Kluwer Academic Publishers. ISSN 0885-6125. URL http://dx.doi.org/10.1023/A:1022633531479.

[40] A. Jain and R. Dubes. *Algorithms for clustering data*. Prentice-Hall, 1988.

[41] T. B. S. Oliveira. Clusterização de dados utilizando técnicas de redes complexas e computação bioinspirada. Master's thesis, Universidade de São Paulo, 2008.

[42] Brian S. Everitt, Sabine Landau, and Morven Leese. *Cluster Analysis*. Wiley, 4th edition, 2009. ISBN 0340761199.

[43] Ricardo Linden. Técnicas de agrupamento, 2009. URL http://www.fsma.edu.br/si/edicao4/FSMA_SI_2009_2_Tutorial.pdf.

[44] Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43:59–69, 1982, Springer-Verlag. ISSN 0340-1200. doi: 10.1007/BF00337288. URL http://dx.doi.org/10.1007/BF00337288.

[45] T. Kohonen. *Self-organization and associative memory: 3rd edition*. Springer-Verlag New York, Inc., New York, NY, USA, 1989. ISBN 0-387-51387-6.

[46] Michael A. Arbib. The handbook of brain theory and neural networks. 2nd ed. Cambridge, MA, 2003.

[47] Pitts W. McCulloch, W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[48] L. Fausett. Fundamentals of neural networks: Architectures, algorithms and applications. *Prentice Hall*, 1994.

[49] L. Xing and D. Pham. Neural networks for identification, prediction, and control. *Springer-Verlag New York, Inc., Secaucus, NJ, USA*, 1995.

[50] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79:2554–2558, 1982, National Academy of Sciences. ISSN 1091-6490.

[51] Teuvo Kohonen. Neurocomputing: foundations of research. chapter Self-organized formation of topologically correct feature maps, pp. 509–521. MIT Press, Cambridge, MA, USA, 1988. ISBN 0-262-01097-6. URL http://dl.acm.org/citation.cfm?id=65669.104428.

[52] T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464 –1480, sep 1990. ISSN 0018-9219. doi: 10.1109/5.58325.

[53] W. J. Freeman. The physiology of perception. *Scientific American*, 1991.

[54] Andreas Bartels and Semir Zeki. The theory of multistage integration in the visual brain. In *Proceedings of the Royal Society*, pp. 2327–2332, 1998.

[55] M. C. Yovits and S. Cameron. Self-organizing systems. *Bookmark*, 1960.

[56] D. J. Willshaw and C. von der Malsburg. How patterned neural connexions can be set up by self-organisation. *Proc Roy Soc B*, 1976.

[57] T. Kohonen. Analysis of a simple self-organizing process. *Biological Cybernetics*, pp. 135–140, 1982.

[58] T. A. Marsland. A review of game-tree pruning. *ICCA Journal*, 1:3–19, 1986.

[59] P. W. Frey. *Chess skill in man and machine /*. Springer-Verlag,, New York :, 1978.

[60] I. Millington. Artificial intelligence for games. *Morgan Kaufmann Publishers Inc.*, 2006.

[61] A. Plaat. *Research Re: search & Re-search*. PhD thesis, Rotterdam, Netherlands, 1996.

[62] Reza Shams and Hermann Kaindl. Using aspiration windows for minimax algorithms. pp. 192–197, 1991, Kaufmann.

[63] V. Manohararajah. Parallel alpha-beta search on shared memory multiprocessors. Master's thesis, University of Toronto, 2001.

[64] M. Gordon Brockington. *Asynchronous Parallel Game-Tree Search*. PhD thesis, Edmonton, Alta., Canada, 1998. AAINQ29023.

[65] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. In *Artificial Intelligence*, pp. 293–326, 1975.

[66] M. G. Brockington. A taxonomy of parallel game tree search algorithms. *Journal of the International Computer Chess Association*, pp. 162–170, 1996.

[67] Valavan Manohararajah. Parallel alpha-beta search on shared memory multiprocessors. Master's thesis, University of Toronto, 2001.

[68] Ian Millington. *Artificial Intelligence for Games (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 0124977820.

[69] J. Schaeffer, M. Hlynka, and V. Jussila. Temporal difference learning applied to a high performance game-playing program. *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 529–534, 2001.

[70] K. Chellapilla and D. B. Fogel. Anaconda defeats hoyle 6-0: A case study competing an evolved checkers program against commercially available software. *Proceedings of the 2000 Congress on Evolutionary Computation CEC00*, pp. 857–863, 2000, California, USA.

[71] K. Chellapilla and D. B. Fotel. Evolving an expert checkers playin program without using human expertise. *IEEE Trans. Evolutionary Computation*, 5(4):422–428, 2001.

[Fierz] M. C. Fierz. Cake informations. http://www.fierz.ch/cake.php (Disponível em 24/05/2013).

[72] Jonathan Schaeffer. Applying the experience of building a high performance search engine for one domain to another, 2002.

[73] A. L. Zobrist. A hashing method with applications for game playing. Technical report, University of Wisconsin, Wisconsin, 1969.

[74] D.M. Breuker, J. W. H. M. Uiterwijk, and H. J. Van Den Herik. Replacement schemes for transposition tables. *ICCA Journal*, 17:183–193, 1994.

[75] D.M. Breuker, J. W. H. M. Uiterwijk, and H. J. Van Den Herik. Information in transposition tables, 1997.

[mpi] The message passing interface (mpi) standard. http://www.mcs.anl.gov/research/projects/mpi/ (Disponível em 24/05/2013).

[76] Gilleanes T. A. Guedes. *UML2: Uma abordagem prática*. Novatec, 2 edition, 2011. ISBN 9788575222812.

[77] H. C. Neto, R. M. S. Julia, and G.S.Caexeta. *LS-Draughts: Using Databases to Treat Endgame Loop in a Hybrid Evolutionary Learning System*. I-Tech Education and Publishing, 2009.

[78] Silva I. N., Spatti D. H., and Flauzino R. A. *Redes Neurais Artificiais: para engenharia e ciência aplicada*. ArtLiber, São Paulo, 2010.