Uma Nova Abordagem de Aprendizagem de Máquina Combinando Elicitação Automática de Casos, Aprendizagem por Reforço e Mineração de Padrões Sequenciais Para Agentes Jogadores de Damas

Henrique de Castro Neto

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2016

Henrique de Castro Neto

Uma Nova Abordagem de Aprendizagem de Máquina Combinando Elicitação Automática de Casos, Aprendizagem por Reforço e Mineração de Padrões Sequenciais Para Agentes Jogadores de Damas

Tese de doutorado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Profa. Dra. Rita Maria da Silva Julia

Uberlândia
2016

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

C355n
2016

Castro Neto, Henrique de, 1981-
Uma nova abordagem de aprendizagem de máquina combinando elicitação automática de casos, aprendizagem por reforço e mineração de padrões sequenciais para agentes jogadores de damas / Henrique de Castro Neto. - 2016.
166 f. : il.

Orientadora: Rita Maria da Silva Julia
Tese (doutorado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Inclui bibliografia.

1. Computação - Teses. 2. Jogo de damas por computador - Teses. 3. Teoria dos jogos - Teses. 4. Aprendizado do computador - Teses. I. Julia, Rita Maria da Silva. II. Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3

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 tese de doutorado intitulada "**Uma Nova Abordagem de Aprendizagem de Máquina Combinando Elicitação Automática de Casos, Aprendizagem por Reforço e Mineração de Padrões Sequenciais Para Agentes Jogadores de Damas**" por **Henrique de Castro Neto** como parte dos requisitos exigidos para a obtenção do título de **Doutor em Ciência da Computação .**

Uberlândia, 21 de Novembro de 2016.

Orientador: _____
Profa. Dra. Rita Maria da Silva Julia
Universidade Federal de Uberlândia

Banca Examinadora:

_____
Prof. Dr. Estevam Rafael Hruschka Júnior
Universidade Federal de São Carlos

_____
Prof. Dr. Luiz Chaimowicz
Universidade Federal de Minas Gerais

_____
Prof. Dr. Carlos Roberto Lopes
Universidade Federal de Uberlândia

_____
Prof. Dr. Marcelo Keese Albertini
Universidade Federal de Uberlândia

Dedico este trabalho à Deus por me conceder saúde e perseverança,
aos meus queridos pais por ter me proporcionado estudo de qualidade,
à minha linda esposa Karina pelo eterno incentivo e a
todos eles pelo amor incondicional e por tudo que representam em minha vida.

Agradecimentos

Aos meus queridos pais Hélio Henrique de Castro e Délia Ferreira de Castro, que com amor e dedicação me proporcionaram estudo de qualidade, em um país onde nem todos têm a mesma oportunidade, para que eu pudesse buscar meu aperfeiçoamento intelectual e profissional: vocês são meu alicerce.

Aos meus irmãos Hélio Júnior e Ana Flávia e ao meu cunhado Hudson de Paula, pelo amor, carinho, convívio e apoio material em alguns momentos difíceis pelos quais passei: obrigado por tudo.

À minha linda esposa Karina Lourenço de Oliveira Castro e às minhas "queridas filhas" Magrela e Futrika, pela eterna alegria, apoio, estímulo e carinho em todos os momentos da minha vida: amo vocês.

Aos meus irmãos de coração Carol Lourenço e Anderson Carvalho por todo o amor e convívio familiar que sempre tivemos, me apoiando em meus projetos e comemorando minhas vitórias: muito obrigado.

À professora Rita Maria da Silva Julia na qualidade de orientadora científica, pelas contribuições para o desenvolvimento deste trabalho, mas também pela confiança e motivação transmitidas. E ainda pelo espírito prático e capacidade de tornar fácil o que parece difícil: minha eterna gratidão.

Aos meus professores da Universidade Federal de Uberlândia que contribuíram para o meu aprendizado, em especial à professora Márcia Aparecida Fernandes e ao professor Paulo Henrique Ribeiro Gabriel, que me ajudaram no direcionamento de meu plano de trabalho: muito obrigado.

Aos grandes amigos e colegas que fiz dentro do doutorado e que compartilharam comigo, direta ou indiretamente, este período de muito trabalho, alegrias e tristezas. Em especial aos colegas e amigos Valquíria Duarte, Clarimundo Júnior, Lídia Bononi e ao secretário da pós-graduação Erisvaldo Araújo. Também quero agradecer ao meu amigo Ernani Melo que viajou comigo para Itália para apresentarmos nossos trabalhos.

Por fim, à FAPEMIG pela bolsa de estudo e suporte financeiro para conclusão deste trabalho e apresentação de artigo em conferência internacional.

"The fact that I bet on myself and didn't lose is a pretty deep reward."
(Jerry Seinfeld)

Resumo

Agentes que operam em ambientes onde as tomadas de decisão precisam levar em conta, além do ambiente, a atuação minimizadora de um oponente (tal como nos jogos), é fundamental que o agente seja dotado da habilidade de, progressivamente, traçar um perfil de seu adversário que o auxilie em seu processo de seleção de ações apropriadas. Entretanto, seria improdutivo construir um agente com um sistema de tomada de decisão baseado apenas na elaboração desse perfil, pois isso impediria o agente de ter uma "identidade própria", o que o deixaria a mercê de seu adversário. Nesta direção, este trabalho propõe um sistema automático jogador de Damas híbrido, chamado *ACE-RL-Checkers*, dotado de um mecanismo dinâmico de tomada de decisões que se adapta ao perfil de seu oponente no decorrer de um jogo. Em tal sistema, o processo de seleção de ações (movimentos) é conduzido por uma composição de *Rede Neural de Perceptron Multicamadas* e *biblioteca de casos*. No caso, a *Rede Neural* representa a "identidade" do agente, ou seja, é um módulo tomador de decisões estático já treinado e que faz uso da técnica de *Aprendizagem por Reforço* TD(λ). Por outro lado, a *biblioteca de casos* representa o módulo tomador de decisões dinâmico do agente que é gerada pela técnica de *Elicitação Automática de Casos* (um tipo particular de *Raciocínio Baseado em Casos*). Essa técnica possui um comportamento exploratório pseudo-aleatório que faz com que a tomada de decisão dinâmica do agente seja guiada, ora pelo perfil de jogo do adversário, ora aleatoriamente. Contudo, ao conceber tal arquitetura, é necessário evitar o seguinte problema: devido às características inerentes à técnica de *Elicitação Automática de Casos*, nas fases iniciais do jogo – em que a quantidade de casos disponíveis na biblioteca é extremamente baixa em função do exíguo conhecimento do perfil do adversário – a frequência de tomadas de decisão aleatórias seria muito elevada, o que comprometeria o desempenho do agente. Para atacar tal problema, este trabalho também propõe incorporar à arquitetura do *ACE-RL-Checkers* um terceiro módulo, composto por uma base de *regras de experiência* extraída a partir de jogos de especialistas humanos, utilizando uma técnica de *Mineração de Padrões Sequenciais*. O objetivo de utilizar tal base é refinar e acelerar a adaptação do agente ao perfil de seu adversário nas fases iniciais dos confrontos entre eles. Resultados

experimentais conduzidos em torneio envolvendo *ACE-RL-Checkers* e outros agentes correlacionados com este trabalho, confirmam a superioridade da arquitetura dinâmica aqui proposta.

Palavras-chave: Teoria dos Jogos. Aprendizagem de Máquina. Aprendizagem por Reforço. Método das Diferenças Temporais. Raciocínio Baseado em Casos. Elicitação Automática de Casos. Mineração de Padrões Sequenciais. Mineração de Dados. Computação Evolutiva. Algoritmo Genético.

Abstract

For those agents that operate in environments where the decision-making needs to take into account, in addition to the environment, the minimizing action of an opponent (such as in games), it is fundamental that the agent has the ability to progressively trace a profile of its adversary that aids it in the process of selecting appropriate actions. However, it would be unsuitable to construct an agent with a decision-making system based on only the elaboration of this profile, as this would prevent the agent from having its "own identity", which would leave it at the mercy of its opponent. Following this direction, this work proposes an automatic hybrid Checkers player, called *ACE-RL-Checkers*, equipped with a dynamic decision-making mechanism, which adapts to the profile of its opponent over the course of the game. In such a system, the action selection process (moves) is conducted through a composition of *Multi-Layer Perceptron Neural Network* and *case library*. In the case, *Neural Network* represents the "identity" of the agent, i.e., it is an already trained static decision-making module and makes use of the *Reinforcement Learning* TD(λ) techniques. On the other hand, the *case library* represents the dynamic decision-making module of the agent, which is generated by the *Automatic Case Elicitation* technique (a particular type of *Case-Based Reasoning*). This technique has a pseudo-random exploratory behavior, which makes the dynamic decision-making on the part of the agent to be directed, either by the game profile of the opponent or randomly. However, when devising such an architecture, it is necessary to avoid the following problem: due to the inherent characteristics of the *Automatic Case Elicitation* technique, in the game initial phases, in which the quantity of available cases in the library is extremely low due to low knowledge content concerning the profile of the adversary, the decision-making frequency for random decisions is extremely high, which would be detrimental to the performance of the agent. In order to attack this problem, this work also proposes to incorporate onto the *ACE-RL-Checkers* architecture a third module composed of a base of *experience rules*, extracted from games played by human experts, using a *Sequential Pattern Mining* technique. The objective behind using such a base is to refine and accelerate the adaptation of the agent to the profile of its opponent in the initial

phases of their confrontations. Experimental results conducted in tournaments involving *ACE-RL-Checkers* and other agents correlated with this work, confirm the superiority of the dynamic architecture proposed herein.

Keywords: Game Theory. Machine Learning. Reinforcement Learning. Temporal Difference Methods. Case-Based Reasoning. Automatic Case Elicitation. Sequential Pattern Mining. Data Mining. Evolutionary Computation. Genetic Algorithm.

Lista de ilustrações

Lista de tabelas

Lista de siglas

AA *Aprendizagem Automática*

ACE *Automatic Case Elicition*

AG *Algoritmo Genético*

AM *Aprendizagem de Máquina*

AR *Aprendizagem por Reforço*

AS *Aprendizagem Supervisionada*

CBSS *Consecutive Binary SubSequences*

DB *endgame DataBase*

DM *Decaying Memory*

EBL *Experience-Based Learning*

EAC *Elicitação Automática de Casos*

GPM *General Positive Memory*

IA *Inteligência Artificial*

ID *Iterative Deepening*

LS *Learning System*

MD *Mineração de Dados*

MLP *Multi-Layer Perceptron*

MCTS *Monte Carlo Tree Search*

MPS *Mineração de Padrões Sequenciais*

MTD *Memory-enhanced Test Driver*

PDN *Portable Draughts Notation*

PG *Programação Genética*

RBC *Raciocínio Baseado em Casos*

RL *Reinforcement Learning*

RNA *Rede Neural Artificial*

SRBC *Sistema de Raciocínio Baseado em Casos*

TD *Temporal Differences*

TT *Tabela de Transposição*

UCT *Upper Confidence bounds applied to Trees*

UCM *Upper Confidence Memory*

Sumário

Capítulo 1

Introdução

1.1 Contextualização

A Teoria dos Jogos tornou-se um importante ramo da Matemática e da Computação na metade do último século, especialmente depois da publicação do clássico livro *"The Theory of Games and Economic Behavior"* [8], uma vez que averigua estratégias apropriadas 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.

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, chegando mesmo a superar os melhores dentre eles. Um dos primeiros trabalhos a obter resultados nessa área de construção de jogadores automáticos foi o jogador de Xadrez de Claude Shannon em 1950 [9]. Nesse trabalho o autor define dois tipos de estratégia em jogos que ainda são muito utilizados nos dias atuais, podendo ser estendidos para outros domínios: uso de *busca exaustiva* para explorar todo o espaço de busca e uso de conhecimento específico ao domínio para examinar apenas uma parte do espaço das jogadas possíveis [10].

Com relação ao espaço de busca em jogos, Van Den Herik, em [4], faz uma análise exaustiva das principais características dos jogos que mais influenciam em sua complexidade. Em particular, são definidas duas medidas de complexidade em jogos: a *complexidade do espaço de estados* e a *complexidade da árvore de jogo*. A complexidade do espaço de estados é definida como o número de posições de jogo legais que podem ser atingidas a partir da posição inicial do jogo. A complexidade da árvore de jogo é definida como o número de nós folha da árvore de busca da solução do jogo a partir da posição inicial do jogo, podendo, inclusive, também ser avaliada através do fator de ramificação médio da árvore de busca para uma determinada profundidade. A principal conclusão de Van Den Herik é a de que, em jogos, a complexidade do espaço de estados é mais relevante do que a complexidade da árvore de jogo como fator determinante para definir a complexidade associada a um jogo.

A tabela 1, que foi extraída e compilada a partir de [3], [4], [5], compara o fator de ramificação e o espaço de estados de alguns jogos de complexidade significante.

Tabela 1 – Complexidade no espaço de busca de alguns jogos [3], [4], [5].

Jogo	**R**amificação	**E**stados
Xadrez	$30 - 40$	10^{46}
Damas 8×8	$8 - 10$	10^{21}
Gamão	± 420	10^{20}
Othello	± 5	$< 10^{28}$
Go 19×19	± 360	10^{172}
Shogi	± 80	10^{71}
Abalone	± 80	$< 10^{29}$

Devido à elevada complexidade no espaço de estados de alguns jogos, como, por exemplo, Damas, Xadrez, Go e Othelo, projetos de construção de agentes jogadores automáticos capazes de jogar no nível de peritos humanos requerem a utilização de complexas e eficientes técnicas de aprendizagem, bem como habilidades específicas que também são demandadas por outros domínios de aplicação, a saber [11], [12]:

- ❏ Aprender a comportar-se em um ambiente onde o conhecimento adquirido é armazenado em uma função de avaliação;
- ❏ Escolha de um número mínimo de atributos possíveis que melhor caracterizem o domínio e que sirvam como um meio pelo qual a função de avaliação adquirirá novos conhecimentos;
- ❏ Seleção da melhor ação para 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;
- ❏ Necessidade de poderosas estratégias de aprendizagem que facilitem a geração de um agente com ótimo nível de desempenho.

Além disso, Arthur Samuel também observou em seus dois principais trabalhos, [13] e [14], que os jogos são, de uma forma geral, um domínio apropriado para o estudo das principais técnicas de *Aprendizagem Automática* (AA), uma vez que, neles, muitas das complicações que surgem nos problemas da vida real são simplificadas, permitindo que os investigadores concentrem-se nos problemas de aprendizagem propriamente dito. Levando em consideração esse fato e a conclusão de Herik [4] que estabelece que a complexidade do espaço de estados é um fator determinante para estimar o nível de dificuldade associado a um jogo, Damas é, portanto, um domínio bastante apropriado para testar e avaliar a eficiência de diferentes tipos de técnicas de *Aprendizagem de Máquina* (AM) e

Inteligência Artificial (IA). Um exemplo de aplicação é o projeto de construção do famoso agente artificial *Chinook* que faz uso de técnicas de *IA* para resolver o problema de Damas. Ele foi projetado em 1989 com o objetivo de derrotar o campeão de Damas mundial da época – Marion Tinsley [15] – e 5 anos mais tarde, in 1994, tornou-se o primeiro campeão de Damas homem-máquina mundial. A arquitetura de *Chinook* conta com um conjunto de funções de avaliação ajustadas manualmente, base de dados de início e fim de jogo (*opening books* e *endgames*), bem como busca *Alfa-Beta* distribuída com *Tabela de Transposição* (TT) e *Aprofundamento Iterativo* – do inglês *Iterative Deepening* (ID) – para escolher a melhor ação a ser executada. Em 2007, a equipe do *Chinook* anunciou que o jogo de Damas está fracamente resolvido (do inglês *weakly solved*), isto é, 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 de que o programa nunca perde, isto é, o programa pode alcançar o empate contra qualquer oponente jogando tanto com as peças pretas quanto as brancas do tabuleiro [16].

Apesar do problema de Damas estar parcialmente resolvido desde 2007 e de *Chinook* ser considerado o jogador artificial de Damas mais forte da literatura [16], ainda existem lacunas abertas no campo da *AM* e *IA* em relação aos estudos e aplicações de diferentes tipos de técnicas de aprendizagem não supervisionada a tal domínio. Uma prova disso é que existem vários jogadores automáticos de Damas fundamentados nas mais diversas técnicas inteligentes não supervisionadas – alguns desses trabalhos, inclusive, foram publicados após o anúncio de que Damas está parcialmente resolvido [12], [17], [18] [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33] e [34]. *Chinook*, ao contrário disso, faz uso de técnicas altamente supervisionadas. Fogel observou isso em [17] ao citar que existe excessiva perícia humana na concepção do campeão de Damas mundial *Chinook*. Esse comportamento supervisionado não é inerente apenas à arquitetura do *Chinook*, outros jogadores automáticos de Damas que são considerados bastante competitivos na literatura, como por exemplo *Cake* e *KingsRow*, também fazem uso de técnicas altamente supervisionadas [35], [36]. Esse é o motivo pelo qual o campo de pesquisa em construção de arquiteturas não supervisionadas tem mostrado ser bastante promissor e crescente nos últimos anos. É justamente dentro desse contexto que o presente trabalho enquadra-se e propõe a construção do sistema híbrido *ACE-RL-Checkers*.

1.2 Motivação

No contexto de jogos de Damas, a equipe na qual o presente trabalho insere-se possui uma série de bem sucedidos jogadores automáticos de Damas não supervisionados [12], [24], [26], [37], [38], [39], [32] e [34]. O trabalho contínuo dos autores para melhorar seus agentes jogadores é motivado pela intenção de estudar, propor e testar ferramentas apropriadas para tratar problemas relativos ao aprendizado de máquina. Tais agentes

tiveram como ponto de partida o jogador automático *NeuroDraughts* de Mark Lynch [6], [40] e baseiam-se em várias técnicas de *AM* e *IA*. No sentido de dar continuidade a esta linha de pesquisa e melhorar a performance dos jogadores obtidos pela equipe, o presente trabalho visa atacar um dos problemas da *IA* que tem sido, ultimamente, foco de pesquisa em aplicações modernas como jogos de entretenimento, *eCommerce*, *eLearning*, robótica, tecnologias assistivas, entre outras [41]. Trata-se do problema de construção de modelos preditivos que possam prever comportamentos futuros de outros agentes tomadores de decisão. Esse problema é conhecido como *modelagem de oponente* (do inglês *opponent modeling*). Atualmente, aplicações tão diversas exigem cada vez mais modelos precisos de outros agentes, sejam artificiais ou humanos [41]. Como exemplos de aplicações que atacam esse tipo de problema, podem-se citar os seguintes:

- *Tecnologias assistivas que proporcionam habilidades funcionais para pessoas com deficiência física*: em [42], Gómez-Sebastià implementa diversas abordagens que tentam mapear padrões comportamentais esperados por atores externos (por exemplo, parentes e/ou enfermeiros) que interagem com pessoas com deficiência física. O foco do trabalho é, portanto, criar tecnologias assistivas que facilitem a interação do dia-a-dia entre esses portadores com deficiência física e as pessoas que pertencem às suas redes sociais (entendendo como é essa relação entre os agentes envolvidos);

- *Navegação autônoma*: em [43], Hussein Dia constrói um agente automático que tenta modelar comportamentos de condutores de veículo em um trânsito. O objetivo é criar um modelo de decisão de rota simples que possa ser ajustado de acordo com os padrões comportamentais de viagem de cada condutor. Em [44] Ali Bazghandi apresenta uma abordagem baseada em modelagem de agente para simulação de fluxo de tráfego de veículos;

- *Comércio eletrônico*: em [45], Serrano apresenta diversas estratégias que modelam perfis de preferência de compras de usuários, com inserção de mecanismos de privacidade, em ambientes de *eCommerce* baseados em agentes. *Sistemas recomendadores de produtos* também são exemplos de aplicações que buscam modelar perfis de usuários dentro do ambiente do comércio eletrônico. Um exemplo é o trabalho de Barry Smyth, em [46], que utiliza recomendação baseada em caso para sugerir diversos tipos de produtos de acordo com as necessidades e preferências de cada usuário;

- *Ensino a Distância*: em [47], [48] e [49], Garrido constrõe diversos tipos de agentes automáticos que auxiliam professores a escolherem as melhores rotas de aprendizagem a distância de acordo com o perfil de cada estudante e os objetivos de cada curso ministrado;

- *Jogos de Entretenimento*: em [50], os autores incorporam diversas estatísticas do jogo Super Mario Bros em modelos de satisfação (preferência) de jogadores para

gerar, em tempo real, diversos cenários personalizados por perfil de jogador. Em [41], Nolan Bard e Michael Bowling modelam estratégias de apostas de oponentes no jogo do Pôquer, usando técnica de filtro de partículas, para aprimorar as apostas de seu agente jogador. Uma outra aplicação interessante, que faz uso de *modelagem de oponente*, é o trabalho de Tetske Avontuur em [51]. Nele os autores usam modelos de perfis de jogadores de Starcraft II, baseados em seus níveis de habilidades, para melhorar as estratégias de ataque e defesa em determinados confrontos do jogo;

Observe que, com base nos exemplos de aplicações citados acima, se um agente for capaz de criar um modelo previsional (ou modelo preditivo) do comportamento de outro(s) agente(s) que interage(m) com ele, então uma resposta mais eficaz pode ser planejada e executada. Este é o foco de pesquisa do presente trabalho: mapear automaticamente e implicitamente o perfil de jogadas realizadas por jogadores de Damas de forma a planejar, com melhor eficiência, futuras tomadas de decisões (movimentos sobre o tabuleiro) de acordo com a dinâmica corrente de jogo de cada oponente.

Em [7], os autores definem uma taxonomia para *modelagem de oponentes* que consiste, basicamente, em categorizar diversos tipos de implementação de modelos de agentes automáticos e listam uma série de técnicas aplicáveis, tais como função de avaliação, redes neurais, modelos baseados em regras, máquinas de estado finito, modelos probabilísticos, modelos baseado em casos, algoritmos genéticos, programação genética e busca Monte Carlo [7]. Além dessas técnicas, alguns modelos de *aprendizagem profunda* (do inglês *deep learning*) também têm sido propostos, recentemente, para atacar o problema de *modelagem de oponente* [52], [53], [54], [55]. Dentre essas técnicas, uma que se destaca por ser aplicável em domínios onde problemas não são completamente compreendidos e onde especialistas expressam seu conhecimento através de exemplos, é a técnica de aprendizagem baseada em experiência chamada *Raciocínio Baseado em Casos* (RBC) [20], [21]. Paralelamente, um exemplo de problema em que o domínio não é totalmente compreendido é o próprio jogo de Damas. Powell cita, em [20], que Damas é um domínio adequado para aplicação da técnica *RBC* devido à sua natureza "não determinística" de seu tabuleiro de jogo. O termo "não determinístico" utilizado por Powell em [20] refere-se ao fato de que um determinado estado de tabuleiro do jogo nem sempre é mapeado para uma única ação "correta" correspondente, isto é, podem haver várias ações "corretas".

Além disso, o próprio problema de *modelagem de oponente* a ser atacado neste trabalho também é caracterizado por ser um domínio não completamente compreendido, uma vez que perfis de jogadas realizadas por jogadores de Damas (aqui representados por sequências de movimentos executados sobre o tabuleiro) nem sempre são explicitamente conhecidos ou completamente observáveis, podendo ser, contudo, extraídos através de exemplos ou experiências vivenciadas por tais jogadores. Em jogos, esse tipo de estratégia é muito comum com humanos. A maioria deles, quando está aprendendo a jogar, só melhora seu nível de jogo estudando boas jogadas realizadas por grandes especialistas, seja

jogando contra eles ou estudando livros/tutoriais com jogadas de mestre. Recentemente, pesquisadores têm descoberto que registros de jogos de especialistas humanos (ou *log de jogos*) cobrem uma área do conhecimento humano muito importante. Devido a isso, técnicas de *Mineração de Dados* (MD) têm sido aplicadas a tais registros de modo a recuperar informações que contribuam para melhorar o nível de inteligência de jogadores automáticos, apesar de existirem poucos trabalhos sobre este tema [2], [56], [57], [58], [59], [60].

Por outro lado, é importante destacar que grandes jogadores automáticos que existem na literatura como, por exemplo, Chinook [16] e Cake [35] em Damas, e Deep Blue II [61] em Xadrez, são altamente supervisionados pois utilizam bases de início e fim de jogo, contendo informações perfeitas sobre qual a melhor ação a ser executada em um determinado tabuleiro, combinadas com *busca exaustiva* e grande capacidade de armazenamento/processamento que pode chegar a milhões de posições avaliadas por segundo em comparação com o que um grande especialista humano chega a processar (avaliar) para escolher a sua melhor ação [62]. Por exemplo, Deep Blue II em Xadrez processa 200 milhões de posições por segundo [62], enquanto Cake, em Damas, processa 2 milhões de posições por segundo [35]. Entretanto, em [62], estudos mostram que grandes mestres de Xadrez, por exemplo, raramente buscam mais do que 100 ramificações em qualquer posição. Mas, então, qual é a chave de sucesso dos humanos ? Em termos computacionais, a "força" dos humanos reside na habilidade de realizar "podas" significativas na árvore de busca e avaliar corretamente as posições resultantes. Isso só é possível porque os humanos são capazes de utilizar diversas formas de raciocínio como, por exemplo, indução, dedução, analogia, probabilidade, etc. Sendo assim, ainda existem pesquisas a serem exploradas, na área de *AM* e *IA*, de forma a projetar-se jogadores automáticos capazes de jogar no nível de peritos humanos sem a necessidade de combinarem um grande volume de processamento com uma grande capacidade de armazenamento.

Além disso, Flinter e Keane citam em [63] que, apesar de jogos como Damas e Xadrez terem sidos "dominados" por agentes que utilizam técnicas computacionais que combinam *busca exaustiva* com grande capacidade de processamento/armazenamento, o mesmo não se pode dizer sobre os agentes para jogos como Go e Shogi, por exemplo, onde o espaço de estados é bastante elevado. A falta de eficiência de tais técnicas supervisionadas para jogos com espaço de busca elevado mostra que esse tipo de abordagem, isto é, *busca exaustiva*, não constitui uma solução apropriada para tais problemas. Nesses casos, uma alternativa é utilizar técnicas inteligentes não supervisionadas ou semi-supervisionadas que possam reduzir o espaço de busca e acelerar a obtenção de uma boa solução [63]. Um exemplo disso é que recentemente o agente *AlphaGo* conseguiu superar o campeão de Go europeu utilizando *redes neurais profundas* (do inglês *deep neural networks*) treinadas através de uma abordagem híbrida que combina *Aprendizagem por Reforço* (AR) – do inglês *Reinforcement Learning* (RL) (um tipo particular de *Aprendizagem não Supervisionada*) – com *Aprendizagem Supervisionada* (AS). Tal feito nunca tinha sido alcançado

até então [64].

Baseado nas evidências apresentadas até aqui e seguindo a vertente de explorar novas arquiteturas não supervisionadas para o domínio de Damas, o presente trabalho propõe uma nova abordagem híbrida que combina as técnicas de aprendizagem de máquina *Elicitação Automática de Casos* (EAC) – do inglês *Automatic Case Elicition* (ACE) (um tipo particular de *RBC*) – e *AR* para ser aplicada em agentes jogadores de Damas. Tal combinação visa criar uma arquitetura híbrida, chamada *ACE-RL-Checkers*, que provê as seguintes contribuições em relação aos agentes que utilizam cada uma dessas técnicas isoladamente: aprimora a exploração aleatória realizada pelos agentes baseados em *EAC*, de forma a introduzir um certo controle na exploração aleatória de tal técnica, além de direcionar a escolha de movimentos para regiões mais promissoras no espaço de busca; por outro lado, introduz adaptabilidade de tomada de decisão nos agentes baseados em *AR* no sentido de escolher as ações (movimentos sobre o tabuleiro) em função da dinâmica de jogo de seus adversários. Em tal dinâmica, o perfil de jogadas do oponente é traçado automaticamente pela técnica *EAC* ao longo das partidas disputadas. Contudo, ao conceber tal arquitetura, é necessário evitar a seguinte fragilidade: nas fases iniciais do jogo, em que a quantidade de casos disponíveis na *biblioteca* da técnica *EAC* é extremamente baixa em função do exíguo conhecimento do perfil de jogo do oponente, a frequência de tomadas de decisão aleatórias é elevada, o que compromete o desempenho do agente. Tal fragilidade ocorre devido a dois motivos: primeiro, porque a *biblioteca de casos* é sempre inicializada (zerada) para cada oponente com o qual o agente interage, isto é, o agente começa jogando sem nenhum conhecimento sobre seu adversário; segundo, devido às características inerentes à técnica *EAC*, a tomada de decisão dinâmica do agente é guiada, ora pelo perfil do adversário – que é quando *casos* são recuperados da *biblioteca*, ora aleatoriamente – que é quando a técnica *EAC* não recupera nenhum caso da *biblioteca*. Essa última situação ocorre em função do próprio mecanismo *EAC* de seleção pseudo-aleatória de casos optar por explorar novas regiões no espaço de busca ou em função da *biblioteca* não possuir informações suficientes reunidas sobre um determinado perfil de jogo do adversário (situação em que há escassez de casos). Veja que esse último caso é bastante comum em *sistemas de recomendação* e é conhecido como *problema de "arranque a frio"* (do inglês *cold-start problem*). Esse problema ocorre quando um recomendador (ou agente) é incapaz de fazer recomendações significativas devido à falta inicial de classificação (ou avaliação) de itens, tais como filmes, músicas, livros, notícias, imagens, páginas web, etc, que são suscetíveis de interesse para um determinado perfil de cliente (usuário) [65], [66], [67], [68]. Neste sentido, visando atacar tal fragilidade, o presente trabalho propõe a inserção de um novo módulo na arquitetura do *ACE-RL-Checkers* composto por uma base de *regras de experiência* mineradas a partir de uma técnica de *Mineração de Padrões Sequenciais* (MPS), um tipo particular de *MD*, aplicada a registros de jogos de especialistas humanos. O objetivo, no caso, é utilizar tais *regras de experiência* nas fases

iniciais do jogo de Damas, de forma a refinar e acelerar o processo de adaptação dinâmica do agente ao perfil de jogo de seu adversário.

1.3 Objetivos e Desafios da Pesquisa

O objetivo geral deste trabalho de doutorado é propor um sistema automático jogador de Damas híbrido, chamado *ACE-RL-Checkers* que, além de introduzir flexibilidade de tomada de decisão por meio de um mecanismo que se adapta ao perfil de seu oponente no decorrer de um jogo, lida com a fragilidade do agente associada ao problema do *cold-start* nas fases iniciais do jogo. Para alcançar o objetivo geral proposto aqui, os seguintes objetivos específicos relacionados na sequência devem ser obtidos.

1.3.1 Objetivos Específicos

1. Implementar um jogador de Damas preliminar baseado em *Redes Neurais* e treinado por *AR* que representará a "identidade" do sistema *ACE-RL-Checkers* a ser produzido nesta pesquisa. Tal jogador, chamado *LS-VisionDraughts*, corresponde a uma versão aperfeiçoada que agrega as principais habilidades de dois agentes predecessores: *LS-Draughts* – do inglês *Learning System* (LS) *of Draughts* – implementado em [12] e *VisionDraughts* implementado em [37]. Nesta direção, *LS-VisionDraughts* combina o eficiente módulo de busca do *VisionDraughts*, baseado na versão *fail-soft Alfa-Beta* com TT e aprofundamento iterativo, com a seleção automática de *features* (funções que descrevem qualitativamente o tabuleiro de Damas) do *LS-Draughts*, baseada no *Algoritmo Genético* (AG), a qual permite uma melhor representação dos estados de tabuleiro na camada de entrada da *RN*. O objetivo, portanto, é criar um agente eficiente, tanto em performance (taxa de vitórias em torneios realizados) quanto tempo gasto para selecionar a melhor ação a ser executada em um determinado tabuleiro, quando comparado com as versões predecessoras que motivaram sua construção. Para tanto, *LS-VisionDraughts* deverá ser avaliado em relação aos seguintes aspectos: desempenho nos torneios, tempo de treinamento, tempo de busca pelo melhor movimento e coincidência de raciocínio durante a escolha de movimentos quando comparado com o raciocínio de um agente fortemente supervisionado na mesma situação;

2. Reimplementar o agente *CHEBR* de Powell [21] que utiliza apenas a técnica de aprendizagem baseada em experiência *EAC* para aprender a jogar Damas. O objetivo é introduzir uma abordagem não determinística de tomada de decisão que se adapta ao perfil de seu oponente no decorrer de um jogo. Em seguida, *CHEBR* deverá ser submetido a um torneio de testes contra o agente *LS-VisionDraughts* com o propósito de testar a eficiência de ambas arquiteturas: uma estática treinada

unicamente por *AR* (*LS-VisionDraughts*) e outra dinâmica treinada unicamente por *EAC* (*CHEBR*);

3. Propor e implementar o sistema híbrido *ACE-RL-Checkers*: arquitetura híbrida que, combinando os módulos *AR* e *EAC*, agrega as habilidades de ambas as abordagens, ao mesmo tempo em que elimina as suas fragilidades. Mais especificamente, *ACE-RL-Checkers* usará o conhecimento provido pela *RN* do *LS-VisionDraughts* para introduzir um certo controle na exploração aleatória da técnica *EAC*, de forma a direcionar a escolha de movimentos para regiões mais promissoras no espaço de busca. Por outro lado, a nova dinâmica aleatória do *ACE-RL-Checkers* norteada por conhecimento introduzirá adaptabilidade ao agente, uma vez que as tomadas de decisão não serão mais determinísticas e as escolhas de movimento do agente serão baseadas na dinâmica corrente de cada jogo de seus adversários. Tal arquitetura híbrida deverá ser avaliada em relação às duas arquiteturas investigadas aqui: arquitetura estática do agente *LS-VisionDraughts* e a arquitetura dinâmica do agente *CHEBR*;

4. Implementar novas estratégias para cálculo do *rating* de forma a melhorar a acurácia dos casos gerados no contexto da técnica *EAC*, fato que permite reduzir a execução de movimentos aleatórios e a quantidade de casos armazenados em memória. As estratégias a serem avaliadas aqui são: estratégia da *Memória Positiva Geral* – do inglês *General Positive Memory* (GPM) – proposta por Powell em [20] e a estratégia da *Memória de Confiança Superior* – do inglês *Upper Confidence Memory* (UCM) – proposta neste trabalho como uma adaptação da técnica *Upper Confidence bounds applied to Trees* (UCT) utilizada com sucesso em grandes jogadores automáticos não supervisionados de Go [69];

5. Propor e implementar um módulo que utiliza uma técnica *MPS* para gerar uma base de *regras de experiência* mineradas a partir de padrões sequenciais de movimentos executados por especialistas de Damas. Tal módulo deverá ser incorporado à arquitetura do sistema *ACE-RL-Checkers* e tem como objetivo lidar com a seguinte fragilidade do agente: nas fases iniciais do jogo em que a quantidade de casos disponíveis na *biblioteca EAC* é extremamente baixa em função do pouco conhecimento sobre o perfil do adversário, a tomada de decisão dinâmica do agente é geralmente comprometida pela alta frequência de execução de movimentos aleatórios sobre o tabuleiro. A nova versão estendida proposta aqui deverá ser avaliada em relação à versão preliminar do *ACE-RL-Checkers* (isto é, versão que não incorpora o *módulo MPS*) nos seguintes aspectos: desempenho no inicio do jogo – medida através da taxa média de movimentos iniciais coincidentes com relação ao forte agente supervisionado *Cake*, o qual faz uso de *bases de início de jogo* (*opening books*) com aproximadamente 2 milhões de movimentos iniciais, tempo de treinamento e melhora

na acurácia dos casos gerados pela técnica *EAC* – medida através dos indicadores quantidade total de movimentos aleatórios gerados pela técnica *EAC* e a quantidade total de casos armazenados na *biblioteca*.

1.4 Hipótese

Considerando os objetivos da pesquisa apresentados na seção 1.3, são hipóteses deste trabalho:

1. A unificação das melhores técnicas dos agentes *LS-Draughts* e *VisionDraughts*, isto é, seleção automática de *features* por meio do *AG* e busca eficiente por meio da versão *fail-soft Alfa-Beta*, permite uma melhor representação dos estados de tabuleiro na camada de entrada da *RN*, além de aumentar o aprofundamento na estrutura da árvore do jogo em busca do melhor movimento. Tal combinação é capaz de produzir um agente eficiente, tanto em performance (taxa de vitórias em torneios realizados) quanto em tempo gasto para selecionar a melhor ação a ser executada em um determinado tabuleiro, quando comparado com as versões predecessoras;

2. Os resultados obtidos por Powell em [21] com a abordagem probabilística de tomada de decisão mostra que a técnica *EAC* é bastante promissora na linha de construção de agentes adaptáveis com a dinâmica de jogadas de seus adversários. Neste sentido, com a implementação de uma abordagem híbrida que utiliza o conhecimento de um agente estático treinado por *AR* para direcionar a exploração pseudo-aleatória da técnica *EAC*, é esperado obter um agente mais adaptável e com melhor desempenho em relação aos agentes que utilizam cada uma dessas técnicas isoladamente: *EAC* ou *AR*;

3. É esperado que características ou jogadas peculiares referentes a um determinado perfil de oponente de Damas sejam descobertas implicitamente através de uma sequência de jogadas (movimentos) e representadas na forma de casos por uma técnica *RBC*;

4. O uso de uma base de *regras de experiência*, mineradas a partir de padrões sequenciais de movimentos executados por especialistas de Damas, pode melhorar a tomada de decisão dinâmica de agentes dotados com a habilidade de adaptação ao perfil de seu oponente, especialmente nas fases iniciais do jogo que é quando o desempenho do agente é prejudicado pela escassez de conhecimento em relação ao perfil de jogo de seu adversário. Sendo assim, a qualidade das regras está diretamente associada à qualidade dos dados (jogos) armazenados em arquivos.

1.5 Contribuições Científicas

As principais contribuições do presente trabalho são:

1. A obtenção de uma eficiente plataforma jogadora de Damas baseada apenas em *Aprendizagem por Reforço*: o *LS-VisionDraughts*;

2. A obtenção de uma nova abordagem híbrida não supervisionada que combina as técnicas de aprendizagem de máquina *AR* e *EAC* para ser aplicada em agentes jogadores automáticos. Tal abordagem utiliza o conhecimento provido por um agente baseado em *AR* para introduzir um certo controle na exploração aleatória da técnica *EAC*, de forma a direcionar a escolha de movimentos para regiões mais promissoras no espaço de busca, fato que refina a qualidade das tomadas de decisão do agente. Além disso, como segunda contribuição, a nova dinâmica aleatória da abordagem híbrida norteada por conhecimento introduz adaptabilidade ao agente, uma vez que as tomadas de decisão não serão mais determinísticas e as escolhas de movimento do agente serão baseadas na dinâmica corrente de cada jogo. Em tal dinâmica, o perfil de jogadas do oponente é traçado automaticamente ao longo das partidas disputadas;

3. A obtenção de novas estratégias para cálculo do *rating* de forma a melhorar a acurácia dos casos gerados no contexto da técnica *EAC*, fato que permite reduzir a execução de movimentos aleatórios e a quantidade de casos armazenados em memória;

4. A obtenção de uma nova abordagem baseada em *MPS* para tratar o problema do *cold-start* em agentes que são dotados da habilidade de, progressivamente, traçar um perfil de seu adversário ao longo dos jogos. Particularmente, nas fases iniciais do jogo em que as informações referentes a esse perfil é ainda escasso, a nova abordagem utiliza *regras de experiência* para prover conhecimento de experiência humana à dinâmica pseudo-aleatória da técnica *EAC* de forma a acelerar o processo de adaptação do agente ao perfil de seu adversário. Tais regras são extraídas a partir de registros de jogos contendo sequências de movimentos de especialistas humanos.

É importante destacar que as metodologias envolvidas nessas contribuições podem ser estendidas para outros jogos de soma zero, isto é, jogos em que o ganho (ou perda) de um dos jogadores é equilibrado, exatamente, pela(s) perda(s), ou ganho(s), do(s) outro(s) participante(s), de tal modo que a soma do total de ganhos dos participantes subtraída de todas as suas perdas será zero (para mais detalhes sobre jogos de soma zero, veja [70]). Entretanto, para que tais metodologias possam ser aplicadas integralmente a outros jogos, a seguinte restrição deve ser avaliada: o *AG* utilizado na construção do agente

LS-VisionDraughts tem como objetivo apenas selecionar, automaticamente, *features* que possam representar o tabuleiro de Damas na entrada de uma rede *MLP*, ao invés de propor ou criar novas *features* que representam o domínio do jogo. Neste sentido, para que o *AG* possa ser utilizado conforme metodologia proposta no objetivo 1 da seção 1.3.1, é necessário que os jogos de soma zero já tenham, prédefinidas, todas as *features* que representam o domínio (estado) desses jogos.

1.6 Organização da Tese

Além da introdução, esta tese está dividida em mais seis capítulos, organizados da forma como se segue.

No capítulo 2 é apresentada toda a fundamentação teórica utilizada para o desenvolvimento prático deste trabalho. Inicialmente, apresenta-se as técnicas utilizadas pelo agente *LS-VisionDraughts*, isto é, busca *Alfa-Beta*, *Rede Neural de Perceptron Multicamadas*, representação de tabuleiro através do mapeamento *NET-FEATUREMAP*, método das *Diferenças Temporais* e *Algoritmo Genético*. Em seguida, é apresentada as técnicas de *Raciocínio Baseado em Casos* e *Mineração de Padrões Sequenciais* utilizadas pelo agente *ACE-RL-Checkers*.

No capítulo 3 são apresentados os trabalhos correlatos com a presente pesquisa.

No capítulo 4 é apresentada a arquitetura do agente *LS-VisionDraughts*, referente ao primeiro objetivo traçado na seção 1.3.1, e os resultados obtidos em torneios contra as versões predecessoras. Neste capítulo também é apresentado o método estatístico de *Wilcoxon* para validação dos resultados obtidos com tal agente.

No capítulo 5 é apresentada, primeiramente, a arquitetura híbrida do sistema *ACE-RL-Checkers*, referente ao terceiro objetivo traçado na seção 1.3.1. Em seguida são apresentadas as duas novas estratégias implementadas para melhorar a acurácia dos valores dos *ratings* dos casos gerados pela técnica *EAC*, conforme proposta apresentada no quarto objetivo da seção 1.3.1. Por fim, os resultados obtidos com a reimplementação do agente *CHEBR* proposto por Powell, referente ao segundo objetivo da seção 1.3.1, são apresentados e comparados com os resultados obtidos em torneio pelo sistema *ACE-RL-Checkers*.

No capítulo 6 é apresentada a nova arquitetura do sistema *ACE-RL-Checkers* com a inclusão do módulo *MPS* para tratativa do problema de *cold-start*, conforme quinto objetivo traçado na seção 1.3.1. Em seguida, a nova versão é submetida a torneios e os resultados validados através do método estatístico de *Wilcoxon*.

Por fim, o capítulo 7 encerra esta tese apresentando as conclusões, contribuições científicas, produções bibliográficas, limitações encontradas e sugestões de trabalhos futuros.

Capítulo 2

Fundamentação Teórica

2.1 Introdução

Neste capítulo é apresentada toda a fundamentação teórica necessária para compreensão das técnicas e estratégias utilizadas para implementação das arquiteturas dos jogadores automáticos de Damas *LS-VisionDraughts* e *ACE-RL-Checkers*, bem como as regras do jogo de Damas adotadas por este trabalho.

2.2 O Jogo de Damas

Damas é um jogo de tabuleiro onde 2 jogadores tentam imobilizar ou capturar todas as peças de seu adversário. Vence quem atingir tal objetivo primeiro. A versão de Damas utilizada neste trabalho é a inglesa, onde o tabuleiro é composto por 64 casas, alternadamente claras e escuras, dispostas em uma matriz quadrada de 8 linhas e 8 colunas. As linhas oblíquas formadas pelas casas escuras são as diagonais, em um total de 15. A mais longa das diagonais, conhecida como *grande diagonal*, tem ao todo 8 casas e une os dois cantos do tabuleiro. Coloca-se o tabuleiro entre os jogadores, de modo que a grande diagonal comece à esquerda de cada jogador e por consequência, a primeira casa à esquerda de cada jogador é escura. O jogo desenrola-se apenas nas casas escuras (também conhecida como *casas ativas*) e cada jogador começa o jogo com doze peças simples localizadas nas três primeiras linhas mais próximas do seu lado. Assim, de um lado terão 12 peças simples brancas e de outro lado terão 12 peças simples preta. A figura 1 mostra a disposição das peças iniciais de um tabuleiro de Damas 8×8.

Os tópicos a seguir resumem as principais regras do jogo de Damas adotadas pelos jogadores automáticos implementados neste trabalho:

- Uma peça simples movimenta-se em diagonal, sobre as casas escuras livres, para a frente, e uma casa de cada vez;

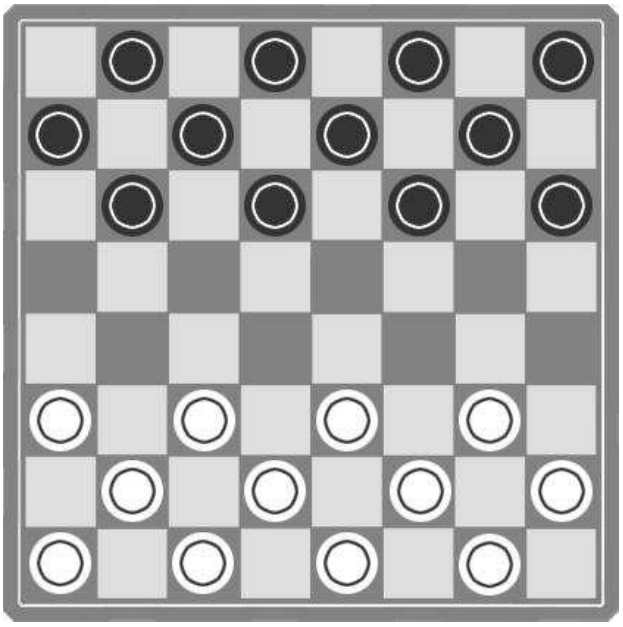

Figura 1 – Configuração inicial de um tabuleiro de Damas 8×8.

- Se uma peça simples atinge o lado oposto do tabuleiro (a última linha), é promovido a *dama* ou *rainha*. Assinala-se a dama sobrepondo, à pedra promovida, outra da mesma cor. A dama é uma peça de movimentos mais amplos que pode tanto andar para frente como para trás em diagonal, entretanto, movimenta-se apenas uma casa de cada vez;

- O jogador pode saltar apenas sobre peças do adversário, nunca sobre as suas próprias peças;

- Se uma peça tem pela frente uma peça adversária e a casa seguinte está livre, então o jogador tem de capturar a peça adversária saltando por cima dela e retirando-a do tabuleiro. Se, após o salto, existirem mais peças adversárias a serem capturadas, o jogador é obrigado a capturar tais peças. É importante destacar que a captura de peça(s) adversária(s) é um movimento obrigatório e tem prioridade em relação a um movimento simples no tabuleiro. A dama pode capturar tanto para frente como para trás, mas uma peça simples pode capturar apenas para frente;

- A versão de Damas implementada neste trabalho não utiliza a *lei da maioria* para captura de peças, isto é, se existir mais de uma opção de captura, não é obrigatório o jogador executar o movimento que capture o maior número de peças. Sendo assim, se o jogador tiver mais de uma possibilidade de capturar peças adversárias pode escolher qualquer uma delas;

- Se o jogador estiver utilizando uma peça simples e optar por uma captura que atinja o lado oposto do tabuleiro (a última linha), tal captura é interrompida nesse ponto

com a promoção da peça simples para damas, isto é, quando uma peça simples é promovida, a jogada sempre termina nessa altura no tabuleiro;

- Duas ou mais peças juntas da mesma cor e na mesma diagonal não podem ser capturadas;

- Na execução de um movimento de captura, é permitido passar mais de uma vez por uma mesma casa vazia;

- Se ambos jogadores executarem os mesmos movimentos em suas 5 últimas jogadas (detecção de *loop* de fim de jogo) ou se o jogo estender para mais do que 100 movimentos, o jogo é declarado empate.

2.3 Agentes Inteligentes

Pode-se definir um *Agente Inteligente* como uma entidade que age em um mundo de acordo com seus objetivos, percepções e o estado atual do seu conhecimento. As ações de um agente são percebidas pela produção de eventos que correspondem às alterações no ambiente em que o mesmo está inserido. Em termos matemáticos, pode-se afirmar que o comportamento do agente é descrito pela função do agente que mapeia qualquer sequência de percepções específica para uma ação [11].

Um agente é uma entidade simples capaz de executar tarefas cuja complexidade varia de acordo com sua construção. Para resolução de problemas mais complexos, faz-se necessária uma interação entre vários agentes, de forma ordenada. Os agentes podem combinar diferentes habilidades para solucionar diferentes problemas [11].

Segundo [71], as propriedades básicas de uma entidade para ser considerada como um agente são:

- **Autonomia:** escolhe uma ação, a ser executada, baseado mais na própria experiência do que no conhecimento recebido inicialmente por seu projetista. Dessa forma, o agente deve aprender o que puder para compensar um conhecimento prévio parcial ou incorreto. As ações do agente não requerem interferência humana direta. Entretanto, pode acontecer situações onde seja necessária a interferência de um agente humano. Mas não projeta-se um agente para ser dependente dessas informações;

- **Reatividade:** reage aos estímulos do ambiente selecionando ações baseadas em sua percepção atual;

- **Proatividade:** capaz de, além de responder a estímulos do ambiente, exibir um comportamento orientado a objetivos;

- **Comunicação:** troca informações com o ambiente e com os outros agentes;

De uma forma geral, os agentes inteligentes são entidades capazes de demonstrar comportamento autônomo, orientado a um objetivo, dentro de um ambiente computacional heterogêneo. A figura 2 mostra a estrutura de um agente inteligente com um *elemento de aprendizagem* e um *elemento de desempenho*. O *elemento de aprendizagem* é responsável pela execução de aperfeiçoamentos do *elemento de desempenho* (ou *função que define o agente*) e esse, por sua vez, é responsável pela seleção de ações externas. O *elemento de aprendizado* utiliza a realimentação do crítico sobre como o agente está funcionando em relação a um padrão fixo de desempenho e determina de que maneira o *elemento de desempenho* deve ser modificado para funcionar melhor no futuro.

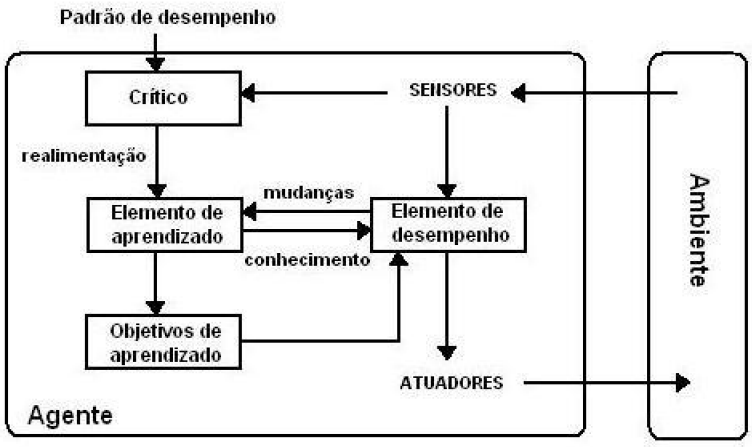

Figura 2 – Modelo geral de um agente inteligente com capacidade de aprendizagem.

Os agentes inteligentes que pretendem ser construídos neste trabalho são jogadores automáticos de Damas que consigam jogar em alto nível de desempenho com praticamente nenhuma intervenção humana. A modelagem desses agentes são discutidas com mais detalhes nos capítulos 4, 5 e 6.

2.4 Estratégia de Busca *Minimax* e a poda *Alfa-Beta*

De forma genérica, as estratégias de busca tradicionais envolvem uma busca em uma árvore que descreve todos os estados possíveis a partir de um determinado estado inicial. Formalmente, o espaço de busca é constituído por um conjunto de nós conectados através de arcos. A cada arco pode ou não estar associado um valor, que corresponde ao custo de transição de um nó a outro. A cada nó é associada uma profundidade, sendo que a mesma tem valor 0 no nó raiz e aumenta de uma unidade para um nó filho. A aridade de um nó é representada pela 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 ao nós já explorados, até a obtenção de uma solução para o problema [27]. O jogo de Damas pode ser visto como uma árvore de possíveis estados de tabuleiros, sendo que a raiz da árvore representa o estado atual do jogo e os nós filhos representam os possível estados de tabuleiros obtidos a partir de movimentos legais.

Entretanto, em problemas onde deseja-se planejar, com antecedência, ações a serem executadas por um agente em um ambiente no qual outros agentes estão fazendo planos contrários àquele, surge o chamado *problema de busca competitiva*. Nestes ambientes as metas dos agentes são mutuamente exclusivas. Os jogos são exemplos de ambientes que apresentam esse tipo de problema de busca competitiva. O jogador não tem que preocupar-se apenas em chegar ao objetivo final, mas também em evitar que algum oponente chegue antes dele, ou seja, vença o jogo. Dessa maneira, o jogador deve antecipar-se à jogada do seu adversário para poder fazer a sua jogada. Uma das maneiras de solucionar esse tipo de problema é através do método de busca *Minimax* [27].

O *Minimax* é uma técnica de busca que determina qual a melhor ação que um agente deve executar, a partir de um determinado estado corrente do jogo, considerando um cenário com dois jogadores, isto é, o *MAX* (o agente) e o *MIN* (o oponente do agente). O algoritmo constrói uma árvore do jogo cuja raiz é a posição corrente do jogo e as folhas são avaliadas pela ótica do jogador *MAX*. Tais valores (avaliação numérica fornecida pela função que define o agente) indicam o quanto os estados correspondentes às folhas são favoráveis para o agente e então, são retropropagados dos nós do nível mais baixo da árvore para o nível mais alto, em direção à raiz, seguindo a seguinte estratégia: os nós do nível minimizar são preenchidos com o menor valor de todos os seus nós filhos e os nós do nível maximizar são preenchidos com o maior valor de todos os seus nós filhos. A figura 3-a) mostra um exemplo de uma árvore do jogo gerada pelo algoritmo *Minimax* na busca pelo melhor movimento para *MAX*. Observe que a *Ação A* é a melhor opção para *MAX*, visto que esse é o movimento que leva para o sucessor de maior avaliação.

Por outro lado, o inconveniente do algoritmo *Minimax* é que ele examina mais estados do que o necessário, o que faz com que o tempo de busca cresça exponencialmente com o aumento da profundidade da árvore do jogo. Tal fato compromete seriamente a performance e o consequente *look-ahead* (maior aprofundamento na estrutura da árvore do jogo) do agente ao longo de uma partida. Uma alternativa para atacar esse problema do algoritmo *Minimax* é o emprego do mecanismo de poda *Alfa-Beta*. O *Alfa-Beta* elimina seções da árvore de busca do *Minimax* que, definitivamente, não podem conter o melhor movimento a ser executado pelo jogador. O algoritmo *Alfa-Beta* pode ser resumido como um procedimento recursivo que escolhe o melhor movimento a ser executado efetuando uma busca em profundidade, da esquerda para a direita, na árvore de busca [72]. O *Alfa-Beta* recebe esse nome devido aos parâmetros *alfa* e *beta* que são passados como argumentos para o algoritmo em conjunto com o estado atual do jogo. *Alfa* e *beta* de-

limitam, respectivamente, o intervalo inferior e superior da janela de busca pelo melhor movimento correspondente ao estado atual do jogo. A avaliação dos nós filhos de um determinado nó no nível de minimização pode ser interrompida quão breve a avaliação de um desses nós seja inferior ao limite mínimo estabelecido (*poda alfa*). Por exemplo, na figura 3-b), o algoritmo *Alfa-Beta* detecta que não é necessário avaliar dois dos nós filhos dos movimentos *Ação B* e *Ação D* (destacados de cinza), visto que os filhos já avaliados possuem valores inferior a janela alfa (0.4). Dessa maneira, o melhor movimento *(Ação A*) é encontrado mais rapidamente do que se fosse procurado pelo algoritmo *Minimax*. Analogamente, a avaliação dos filhos de um nó maximizador pode ser interrompida quão breve a avaliação de um desses nós seja superior ao limite máximo estabelecido (*poda beta*).

O algoritmo *Minimax* com poda *Alfa-Beta* é utilizado neste trabalho pelo agente *LS-VisionDraughts* para selecionar, na árvore de busca do jogo, o melhor movimento que o agente deve executar em função do estado corrente do jogo. A seção 4.5 explica com mais detalhes como o algoritmo *Alfa-Beta* é implementado em tal agente com a adição dos recursos TT e ID que permitem reduzir ainda mais o tempo de busca pelo melhor movimento.

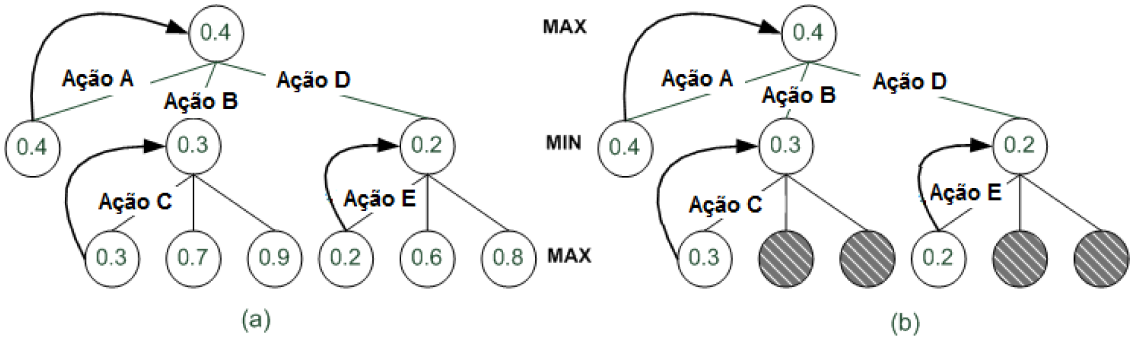

Figura 3 – a) Árvore de busca expandida pelo algoritmo *Minimax*; b) Árvore de busca expandida pelo algoritmo *Alfa-Beta*.

2.5 Rede Neural de Perceptron Multicamadas

Uma *Rede Neural Artificial* (RNA) é um modelo computacional baseado em redes neurais biológicas que são compostas por unidades básicas chamadas *neurônios*. Esses *neurônios* artificiais simulam o funcionamento de uma célula nervosa que é composta basicamente por um corpo celular (*soma*), responsável pelos processos metabólicos da célula, e pelas projeções desse corpo (*dendritos* e o *axônio*), que determinam as conexões sinápticas com outras células cerebrais [73].

O primeiro modelo matemático de um neurônio artificial foi proposto em 1943 por McCulloch and Pitts [74]. Nesse modelo, um neurônio é uma unidade de processamento

que aplica uma *função de ativação* g a um conjunto de n entradas que são combinadas por meio de uma *função de entrada* ou *função in* (impulso nervoso local). Mais especificamente, conforme pode ser visto na figura 4, as entradas $\{x_0, ..., x_n\}$ de um neurônio j são geradas por outras unidades de processamento (ou neurônios) cujas forças de conexões para o neurônio j são representadas pelos pesos $\{w_{0,j}, ..., w_{n,j}\}$. Em outras palavras, $w_{i,j}$ representa a força da conexão entre a unidade de processamento i, que gera a entrada x_i, e o neurônio j.

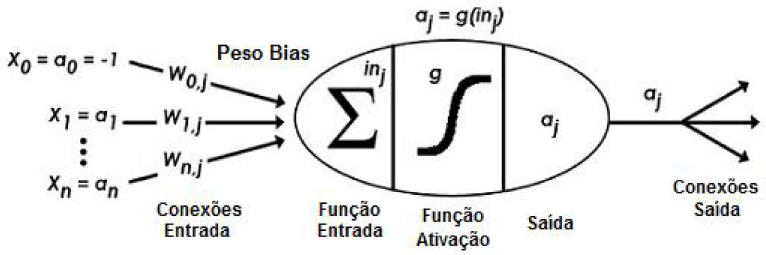

Figura 4 – Modelo de um neurônio artificial

Nessa figura, o *limiar* ou *bias* corresponde a um valor arbitrário representado pelo sinal de entrada x_0 que é conectado ao neurônio j com peso $w_{0,j}$. Esse valor é utilizado como uma referência que estabelece o limite mínimo que deve ser atingido pelos demais sinais de entrada do neurônio, cujo valor combinado determina o disparo do sinal de saída do neurônio j. Em outras palavras, o neurônio j emite o sinal de saída a_j se, e somente se, a seguinte restrição (1) é satisfeita:

$$\sum_{i=1}^{n} w_{i,j} \cdot x_i > w_{0,j} \cdot x_0. \tag{1}$$

Então, a saída a_j correspondente ao neurônio j da figura 4 pode ser calculada por meio da equação 2:

$$a_j = g(in_j) = g(\sum_{i=0}^{n} w_{i,j} \cdot x_i). \tag{2}$$

Observe que a *função de ativação* g deve ser projetada para atender às necessidades de cada problema. Nesse sentido, existem diversas funções de ativação disponíveis na literatura para cada tipo de problema: funções *limiar*, *linear*, *rampa*, *sigmóide*, *tangente hiperbólica* e outras [73].

Uma RNA é então composta por um conjunto de neurônios interconectados, onde a saída a_i de um neurônio i corresponde à entrada x_i do neurônio j no qual i está conectado com peso $w_{i,j}$. Uma RNA deve ser projetada de tal forma a ser capaz de resolver um determinado problema. Para tanto, ela precisa ser treinada adequadamente para o problema para o qual ela foi projetada. O *processo de treinamento* de uma RNA consiste basicamente em ajustar seus pesos sinápticos até encontrar as combinações adequadas

entre seus neurônios de forma a produzir a saída desejada correspondente ao sinal de entrada. Em outras palavras, o ajuste sináptico entre os neurônios de uma RNA representa o aprendizado em cada neurônio do fato apresentado, isto é, cada neurônio, conjuntamente com todos os outros, representa a informação que atravessou 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 em seus pesos sinápticos [73].

Particularmente, os *Perceptrons de Multicamadas* – do inglês *Multi-Layer Perceptron* (MLP) – são RNAs caracterizadas pela presença de pelo menos uma camada intermediária ou camada oculta de neurônios – camada em que os neurônios são efetivamente unidades processadoras, mas que não correspondem à camada de saída. É importante destacar que, ao contrário dos perceptrons simples, as *MLPs* são capazes de aprender qualquer tipo de problema não-linear [73].

A figura 5 mostra uma exemplo de arquitetura *MLP* com m camadas processadoras. Visando melhorar a notação para a representação de uma saída arbitrária a_i correspondente ao neurônio i pertencente à *l-th* camada de uma *MLP*, a_i será reescrito, deste ponto em diante, como a_i^l. Similarmente, o sinal de saída a_j^m correspondente à saída de um neurônio j pertencente à camada de saída da *MLP* será referenciado como O_j. Para o exemplo apresentado na figura 5, a saída a_j^m correspondente ao *j-th* neurônio da camada de saída é dada pela seguinte equação:

$$O_j = g(in_j) = g(\sum_{i=0}^{n} w_{i,j}^{m-1} \cdot a_i^{m-1}) = a_j^m, \tag{3}$$

onde a_i^{m-1} é a saída do *i-th* neurônio da camada *m-1* conectado ao neurônio de saída j com peso $w_{i,j}^{m-1}$. Observe que a saída a_i^{m-1} também corresponde ao sinal de entrada x_i do neurônio j. Considerando os dois tipos principais de aprendizagem em uma RNA, no aprendizado supervisionado a rede ajusta seus pesos sinápticos de tal forma a minimizar a diferença (também conhecida por *error*) entre o valor O_j obtido e a saída desejada. Essa diferença é então utilizada como parâmetro no cálculo dos ajustes dos pesos da RNA [73]. Já no aprendizado não supervisionado, particularmente no processo de treinamento por *AR*, o valor de saída desejado da rede não é conhecido. Nesse caso, a diferença é substituída pela diferença entre duas saídas sucessivas produzidas pela RNA. Maiores detalhes desse processo são apresentados na seção 4.4.1.

Conforme é apresentado na seção 4.3, o sistema *LS-VisionDraughts* proposto neste trabalho adota uma arquitetura *MLP* com 20 neurônios em uma única camada oculta e 1 neurônio na camada de saída. Tal arquitetura também foi adotada pelas versões predecessoras do *LS-VisionDraughts*: *NeuroDraughts* [40], *LS-Draughts* [12] e *VisionDraughts* [37]. Essa arquitetura foi mantida com o objetivo de criar-se um rastro de comparação contínuo na evolução desses agentes.

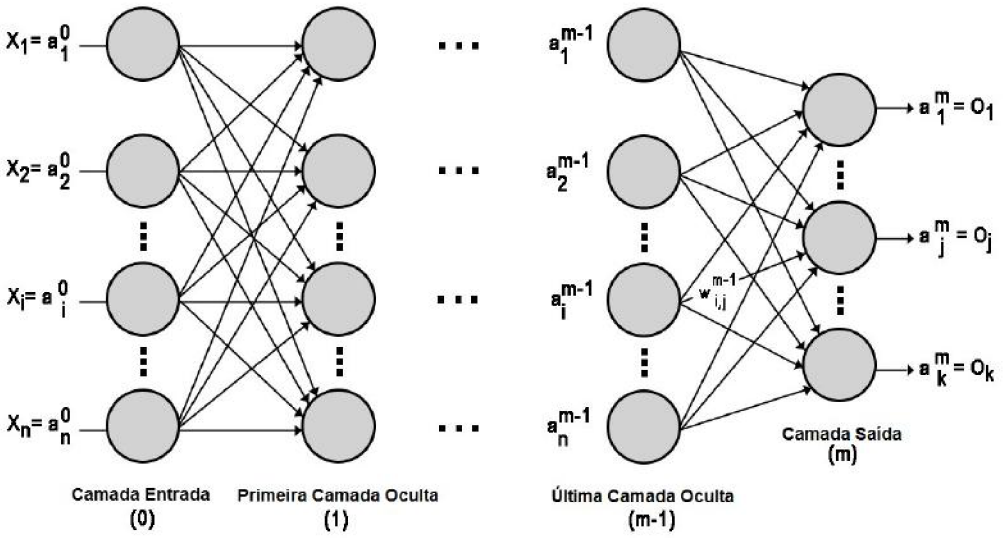

Figura 5 – Arquitetura de uma *MLP* ou *Perceptron de Multicamadas.*

2.6 Mapeamento de Tabuleiro *NET-FEATUREMAP*

A utilização de um conjunto de *features*, também conhecidas como *características* qualitativas referentes a um determinado domínio de aplicação, para representar um tabuleiro de Damas na arquitetura de um agente automático foi primeiramente proposta por Samuel [13] com o intuito de prover medidas numéricas que melhor representam as diversas propriedades de posições de peças sobre um tabuleiro. Várias dessas *features* implementadas por Samuel resultaram de análises feitas sobre o comportamento de especialistas em partidas de Damas. Em termos práticos, essas análises tinham como objetivo tentar descobrir quais *features* referentes a um estado de tabuleiro, tais como, peças em vantagens, quantidade de damas (rainhas) sobre o centro do tabuleiro, quantidade de peças sob ameaça do oponente, e outras, são frequentemente analisadas e selecionadas pelos próprios especialistas quando vão escolher seus movimentos de peças (ações) sobre o tabuleiro durante uma partida de Damas.

Samuel implementou 26 *features* para treinar seu jogador de Damas, cuja função de avaliação era um polinômio. Os termos desse polinômio representavam subconjuntos das 26 *features* e, os coeficientes, os pesos (ou a importância) das *features* para o agente jogador. Para ajustar tais coeficientes, Samuel combinou algumas técnicas heurísticas com AM para treinar e melhorar o desempenho de seu jogador de Damas [13], [14].

O agente automático de Damas de Mark Lynch, *NeuroDraughts*, é um outro exemplo de aplicação que também utiliza um conjunto de *features* para aprender a jogar Damas [40], [6]. Mais especificamente, Lynch define um mapeamento de tabuleiro chamado *NET-FEATUREMAP* como um conjunto de *features* do tipo $f_i : B \longrightarrow \mathbb{F}$, onde B representa o conjunto de todos os tabuleiros possíveis do jogo de Damas e F um conjunto de 12 *features* que representam quantitativamente e qualitativamente um tabuleiro do jogo de

Damas. Cada *feature* tem um valor absoluto que é convertido em uma sequência binária e associado à entrada de uma RNA. A figura 6 mostra um esboço de como funciona o mapeamento de tabuleiro *NET-FEATUREMAP* adotado por Lynch para mapear qualquer tabuleiro de Damas na camada entrada de uma RNA.

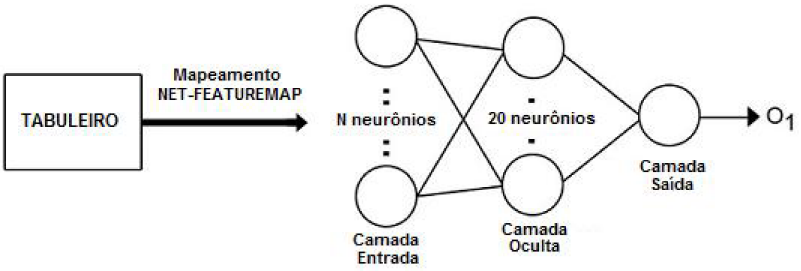

Figura 6 – Esboço do mapeamento de tabuleiro *NET-FEATUREMAP* na entrada de uma RNA.

As 12 *features* implementadas por Lynch no mapeamento *NET-FEATUREMAP* podem ser vistas na tabela 2. Cada *feature* tem um valor absoluto que representa a sua medida analítica sobre um determinado estado de tabuleiro. Esse valor é depois convertido em *bits* significativos que, em conjunto com os demais *bits* das outras *features* presentes no mapeamento, constituirão a entrada na *MLP*. Totalizando a soma de *bits* das *features* listadas na tabela 2, são 38 entradas na *MLP* que foram utilizadas por Lynch para mapear qualquer tabuleiro do jogo de Damas. Para mais detalhes sobre o mapeamento *NET-FEATUREMAP* e de como é realizado a conversão do valor absoluto de cada *feature* em *bits* significados, veja [11].

2.7 Aprendizagem de Máquina Supervisionada x Não Supervisionada

A capacidade de aprender é um dos atributos fundamentais do comportamento de um *agente inteligente* e requer a implementação de um processo multifacetado. Em tal processo, a aprendizagem inclui a aquisição de novos conhecimentos, o desenvolvimento de habilidades motoras e cognitivas através da prática ou de instrução, a organização de novos conhecimentos em representações gerais e efetivas e a descoberta de novos fatos e teorias através da observação e da experimentação. O estudo e modelagem computacional dos processos de aprendizagem em suas múltiplas manifestações constitui o problema da *Aprendizagem de Máquina* [75]. Dentro desse contexto, as tarefas de aprendizado de máquina são tipicamente classificadas em duas amplas categorias:

- *Aprendizagem Supervisionada*: abordagem que trabalha com sistemas que aprendem através de exemplos de pares de entrada e saída. Tais pares fornecem aos sistemas

Tabela 2 – As 12 *features* utilizadas por Lynch no mapeamento *NET-FEATUREMAP* [6]

Features	**Descrição Funcional**	**Bits**
PieceAdvantage	Contagem de peças em vantagem para o jogador preto.	4
PieceDisadvantage	Contagem de peças em desvantagem para o jogador preto.	4
PieceThreat	Total de peças pretas que estão sob ameaça.	3
PieceTake	Total de peças brancas que estão sob ameaça de peças pretas.	3
Advancement	Total de peças pretas que estão na 5ª e 6ª linha do tabuleiro menos as peças que estão na 3ª e 4ª linha.	3
DoubleDiagonal	Total de peças pretas que estão na diagonal dupla do tabuleiro.	4
Backrowbridge	Se existe peças pretas nos quadrados 1 e 3 e se não existem damas brancas no tabuleiro.	1
Centrecontrol	Total de peças pretas no centro do tabuleiro.	3
XCentrecontrol	Total de quadrados no centro do tabuleiro onde tem peças brancas ou que elas podem mover.	3
TotalMobility	Total de quadrados vazios para onde as peças brancas podem mover.	4
Exposure	Total de peças pretas que são rodeadas por quadrados vazios em diagonal.	3
KingCentreControl	Total de damas pretas no centro do tabuleiro.	3

indicativos de como comportar-se para tentar aprender uma determinada função que "poderia" gerá-los. Formalmente, isto significa que, dados exemplos de pares $(x_i, f(x_i))$, onde x_i é a entrada e $f(x_i)$ é a saída da função aplicada a x_i, então a tarefa é encontrar, dentre uma coleção de exemplos de f, uma função h que mais aproxime de f. Esses métodos são apropriados quando existe o auxílio de um "professor" (supervisor) fornecendo os valores corretos para a saída da função de avaliação;

- *Aprendizagem Não Supervisionada*: abordagem que trabalha com sistemas onde não há nenhum "professor" fornecendo exemplos de pares de entrada e saída. Tais sistemas são projetados para aprenderem padrões de conhecimento através de interações do tipo *tentativa-erro* com o ambiente – um exemplo de aplicação é a técnica de *aprendizagem por reforço* que será explicada na seção 2.8 – ou através de extrações automáticas de classes ou rótulos que são similares (comuns) em uma determinada base de dados – um exemplo desse tipo aplicação é a técnica de *mineração de padrões sequenciais* que será explicada na seção 2.11.

Em relação às duas categorias de aprendizagem apresentadas acima, dois pontos merecem ser destacados: 1) Existem projetos de *AM* híbrida que combinam aprendizagem supervisionada com aprendizagem não supervisionada, podendo, inclusive, ser classificada como *aprendizagem semi-supervisionada* [13], [71]; 2) Existem projetos de *AM* supervisionada onde o agente é menos autônomo, isto é, ao invés do agente aprender exemplos de pares de entrada e saída de um determinado domínio, ele simplesmente lê rótulos de entrada e saída com informações perfeitas que o auxilia em seu processo de tomada de decisão em *tempo real*. Os jogadores supervisionados de Damas *Cake* [35] e *Chinook* [15] são exemplos de agentes que adotam esse tipo de estratégia. Como será apresentado na seção 3.2, tais jogadores fazem uso de uma coleção de jogadas (pares de entrada e saída) utilizadas por grandes mestres na fase inicial do jogo, conhecida como *opening books*, e também uma base de dados de final de jogo, conhecida como *endgames*, que contém milhões de estados de tabuleiro do jogo. Tais bases são geradas manualmente, através de rótulos extraídos de jogos de especialistas humanos ou através de algoritmos de *busca exaustiva* que utilizam técnicas de *análise em retrocesso* (mais detalhes sobre tal técnica, veja [76]), ao invés de serem extraídas automaticamente por meio de uma técnica de aprendizagem [15], [17], [35], [77].

2.8 Aprendizagem por Reforço e o Método das Diferenças Temporais TD(λ)

Segundo Sutton e Barto [78], *Aprendizagem por Reforço* nada mais é do que a aplicação dos conceitos básicos de *AM*: um indivíduo deve aprender a partir da sua interação com o ambiente onde ele encontra-se, através do conhecimento do seu próprio estado no ambiente, das ações efetuadas no ambiente e das mudanças de estado que aconteceram depois de efetuadas as ações. A importância de utilizar *AR* como uma técnica de aprendizagem está diretamente ligada ao fato de tentar-se obter uma política próxima à 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 essa política, isto é, a função do *agente-aprendiz* [27].

Dentre os algoritmos existentes para solucionar o problema de *AR*, os métodos das Diferenças Temporais de Sutton – do inglês *Temporal Differences* (TD) ou TD(λ) – se destacam por não exigirem um modelo exato do sistema e por permitirem ser incrementais na busca de soluções para problemas de predições [79]. Aprender a predizer é uma das formas mais básicas e predominantes em aprendizagem. Através de um certo conhecimento, alguém poderia aprender a predizer, por exemplo, considerando o domínio de aplicação deste trabalho, se uma determinada disposição de peças no tabuleiro de Damas conduzirá a uma vitória.

Os métodos TD(λ) são guiados pela *diferença* entre os valores de predições de sucesso temporárias que são associadas aos estados sucessivos experimentados pelo agente, em um dado domínio, em decorrência de uma sequência de ações $\{a_0, ..., a_{i-1}, a_i, a_{i+1}, ...a_j\}$ que ele executa, ao longo do tempo, com o objetivo de realizar uma determinada tarefa f (aqui chamada de *fim-de-episódio*) para o qual ele foi projetado. Deste ponto em diante, quando a sequência de ações produz um *fim-de-episódio* (que pode ser definido por um único estado ou por um conjunto de estados), a última ação a_j da sequência, cuja execução gerou o estado *fim-de-episódio*, será representado por a_f. Por exemplo, considerando um agente jogador, a_f corresponde a uma ação que produz o estado de fim de jogo [27].

Supondo que um agente execute, sobre um determinado domínio, uma sequência de ações $\{a_0, ..., a_{i-1}, a_i, a_{i+1}, ..., a_f\}$ ao longo do tempo, onde a ação a_f produz um *fim-de-episódio*, o processo de aprendizagem de um agente treinado por TD(λ) pode ser resumido da seguinte forma: após a execução da ação a_f, o agente receberá do ambiente um reforço positivo, caso o estado de *fim-de-episódio* produza um resultado (ou medida de desempenho) que seja satisfatório para o objetivo pelo qual o agente foi projetado – por exemplo, no caso de Damas, se o agente vencer – ou receberá uma punição (reforço negativo do ambiente) caso o estado de *fim-de-episódio* produza um resultado não satisfatório para o objetivo pelo qual o agente foi projetado – novamente, no exemplo de Damas, se o agente perder. Resumindo, as predições de *fim-de-episódio*, isto é, P_f, são predições que representam reforços positivos ou negativos retornados pelo próprio ambiente em função do desempenho final do agente no *fim-de-episódio*. Para todos os outros estados anteriores ao *fim-de-episódio*, onde nenhuma recompensa é retornada pelo ambiente, é a própria função de avaliação que define o agente que calcula a predição de sucesso P_i associada ao estado resultante da ação executada a_i, onde $i \in \{0, 1, ..., i-1, i, i+1...., f-1\}$. Assim, para cada duas predições temporais sucessivas P_i, predição associada ao estado corrente, e P_{i+1}, predição associada ao estado sucessivo, de um episódio, o mecanismo TD ajusta a função de avaliação do agente com base na diferença obtida por $(P_{i+1} - P_i)$ – daí a origem do nome do método "Diferenças Temporais". O mesmo ocorre com o *fim-de-episódio* e consequentemente com o reforço (positivo ou negativo) P_f retornado pelo ambiente após a execução da ação a_f. O mecanismo TD usa essa informação retornada pelo ambiente para ajustar a função de avaliação do agente com base na diferença obtida pelas duas predições sucessivas [27].

Observe que esse tipo de comportamento dos métodos TDs tornam-os adequados para serem utilizados em jogos, como Gamão, Xadrez, Damas, entre outros, onde há caracterização de problemas de predição: para cada estado de tabuleiro o agente deverá escolher qual ação a ser executada de forma que o estado resultante tem uma predição que é próxima à possibilidade de vitória.

No sistema *LS-VisionDraughts* detalhado no capítulo 4, as sequências de ações correspondem às sequências de movimentos executados pelo agente ao longo dos jogos e as

predições correspondem às avaliações feita pela rede *MLP* para os tabuleiros envolvidos no processo de busca pelo melhor movimento. Detalhes de como a rede *MLP* do *LS-VisionDraughts* aprende a jogar Damas através dos métodos TDs são apresentados nas seções 4.4.1 e 4.4.2.

2.9 Algoritmo Genético

Dentro do campo de pesquisa em *AM*, uma classe de meta-heurísticas que têm recebido bastante atenção nos últimos anos é composto por técnicas de *Computação Bioinspirada* que adotam metáforas e modelos de sistemas biológicos no desenho de soluções computacionais para problemas complexos [80]. Um dos grupos de algoritmos bioinspirados mais largamente utilizados na literatura são os *Algoritmos Genéticos* introduzidos por John Holland na década de 1960 com o objetivo de estudar formalmente os conceitos de adaptação que ocorrem na natureza, formalizá-los matematicamente e desenvolver sistemas artificiais que mimetizam os mecanismos originais encontrados em sistemas naturais [81].

O *AG* proposto por Holland é um método que consiste em modificar uma população inicial, conjunto de indivíduos representando as soluções candidatas codificadas na forma de cromossomos, em uma nova população utilizando a seleção natural e os operadores genéticos *recombinação gênica* (ou *crossover*) e *mutação*. Um indivíduo da população é representado por um único cromossomo, que contém a codificação (genótipo) de uma possível solução do problema (fenótipo). Cromossomos são geralmente implementados na forma de listas de atributos, vetores ou *arrays*, onde cada atributo é conhecido como gene, e os possíveis valores que um determinado gene pode assumir são denominados *alelos*. No caso particular do *AG* proposto por Holland, um cromossomo é geralmente representado por um vetor binário de genes [27].

Apesar dos *AGs* apresentarem etapas não-determinísticas em seu processo de execução, eles não são métodos de busca puramente aleatórios, pois combinam variações aleatórias com seleção, polarizada pelos valores de adequação da *função de adaptabilidade* ou *fitness* atribuído a cada indivíduo. Os *AGs* possuem um paralelismo implícito decorrente da avaliação independente de cada uma das cadeias de gene que compõem os cromossomos, ou seja, pode-se avaliar a viabilidade de um conjunto de solução para o problema. O processo de busca é, portanto, multi-direcional, com a manutenção de soluções candidatas que representam a busca em várias partes do domínio e com troca de informações entre essas soluções. A cada geração, soluções relativamente "boas" reproduzem-se mais frequentemente, enquanto que soluções relativamente "ruins" tendem a ser eliminadas. Para fazer a distinção entre diferentes soluções é empregada a *função de adaptabilidade* que simula o papel da pressão exercida pelo ambiente sobre o indivíduo. O pseudo-código 1 descreve um *AG* típico [27].

Resumidamente, no desenvolvimento de um *AG* para um determinado problema,

Algoritmo 1 :Pseudo-código do *Algoritmo Genético*

```
1: procedure AG()
2: method:
3:       t ← 0 (geração zero do AG);
4:       Gere aleatoriamente T_p indivíduos da população inicial, isto é, G_0;
5:       Calcule o fitness de G_0;
6: while Critério de Parada não for satisfeito do
7:    t = t + 1 (geração seguinte);
8:    Selecione os pais da geração G_{t-1};
9:    Aplique os operadores de crossover sobre os pares de cromossomos pais, obtendo G_t;
10:   Aplique os operadores de mutação sobre G_t;
11:   Calcule o fitness de G_t;
12:   Atualize G_t selecionando os T_p melhores indivíduos dentre as gerações G_{t-1} and G_t;
13: end while
```

devem-se especificar os seguintes componentes [82], [80]:

- *Representação genética para soluções potenciais*: a codificação de um indivíduo é uma das etapas mais críticas na definição de um *AG*. A definição inadequada da codificação pode acarretar diversos problemas, entre esses, o problema da convergência prematura do *AG*. A convergência prematura ocorre quando indivíduos relativamente adaptados, contudo não ótimos, rapidamente dominam a população fazendo com que o *AG* convirja para um máximo ou mínimo local. Dessa forma, a estrutura de um cromossomo deve representar uma solução como um todo e deve ser a mais simples possível;

- *Procedimento para geração da população inicial*: o método mais conhecido é a geração aleatória dos indivíduos. Se algum conhecimento inicial a respeito do problema estiver disponível, esse pode ser utilizado na inicialização da população;

- *Definição do método de seleção dos indivíduos para a próxima geração*: segundo [82], a seleção desempenha papel fundamental na evolução, já que é o componente do processo evolutivo responsável por determinar os indivíduos "vencedores" e "perdedores" na luta pela sobrevivência. Há vários métodos de seleção de indivíduos para aplicação dos operadores genéticos, dentre esses, dois merecem destaque: o método da *roleta estocástica* e a seleção por *torneio estocástico* [11];

- *Definição dos operadores genéticos com base na codificação utilizada*: o princípio básico dos operadores genéticos é transformar a população, efetuando modificações em seus indivíduos, permitindo a diversificação e manutenção de características de adaptação adquiridas nas gerações anteriores. Os operadores genéticos mais frequentemente utilizados em *AGs* são o *crossover* de um ponto e a mutação. Geralmente, são atribuídos valores pequenos para a taxa de mutação. A idéia por trás do operador de mutação é criar uma variabilidade extra na população, mas sem destruir o progresso já obtido com a busca;

- *Definição da função de adaptabilidade ou fitness*: é importante definir uma função que possa classificar as soluções em termos de sua adaptação ao ambiente, isto é, sua capacidade de resolver o problema;

- *Definição do critério de parada do* AG*, tais como*: a) quando o algoritmo atingir um determinado número de gerações; b) quando for atingido um determinado valor para a *função fitness*, definido *a priori*; c) quando não ocorrer melhora significativa no *fitness* do melhor indivíduo por um determinado número de gerações;

- *Definição de valores para alguns parâmetros do* AG: tamanho da população, probabilidades de aplicação dos operadores genéticos e outros. É importante destacar que o tamanho da população afeta diretamente o desempenho global e a eficiência dos resultados do *AG*. Populações muito pequenas tendem a perder a diversidade genética rapidamente e podem não obter uma boa solução. Por outro lado, se a população for muito grande, o algoritmo tenderá a ser muito caro computacionalmente (lento), principalmente se o cálculo da função de *fitness* for complexo.

O *AG* implementado no sistema *LS-VisionDraughts*, detalhado no capítulo 4, é utilizado para gerar, automaticamente, um conjunto mínimo de *features* do mapeamento *NET-FEATUREMAP*, descrito na seção 2.6, com o objetivo de representar adequadamente os estados do tabuleiro do jogo de Damas na entrada da *MLP* do agente. Esse processo é discutido com detalhes na seção 4.2.

2.10 Raciocínio Baseado em Casos

Raciocíno Baseado em Casos é uma técnica da *IA* com enfoque na solução de problemas através do aprendizado baseado em experiências vivenciadas por um agente. Daí *RBC* também ser conhecido como uma técnica de *Aprendizagem Baseada em Experiência* – do inglês *Experience-Based Learning* (EBL). Em geral, um algoritmo de *RBC* busca a solução para uma situação atual através da recuperação e adaptação de soluções passadas semelhantes, dentro de um mesmo domínio do problema. A recuperação dos dados é realizada em memória, verificando se existem casos semelhantes com as características atuais do problema, podendo encontrar um ou mais casos e adaptá-los de alguma maneira, para que se ajuste ao problema atual, criando um novo caso para uso futuro [83].

Um *Sistema de Raciocínio Baseado em Casos* (SRBC) deve ser capaz de simular a capacidade humana de relembrar e adaptar uma experiência passada na resolução de um novo problema. Para ser capaz de realizar tal função, um *SRBC* deve ser projetado contemplando alguns componentes básicos, tais como [1]:

- *Representação do Conhecimento*: em um *SRBC* o conhecimento geralmente é armazenado na forma de casos dentro de uma *biblioteca de casos*;

- *Medida de similaridade*: é o cálculo feito entre o novo problema inserido no sistema e os casos armazenados na *biblioteca de casos* para identificar quais desses casos serão utilizados como base para resolução do novo problema;
- *Adaptação*: adicionar atributos ao caso mais similar para resolução do novo problema;
- *Aprendizado*: armazenamento na *biblioteca de casos* dos novos casos resolvidos com sucesso.

A *representação do conhecimento* é a parte mais importante de um sistema *RBC* e deve conter atributos chaves para serem utilizados no cálculo de similaridade entre os novos casos e aqueles já armazenados na *biblioteca de casos*. Um caso pode ser representado como um objeto dentro do sistema de forma que os atributos daquele objeto são padrões dentro de um determinado contexto. A figura 7 ilustra como pode ser feita a representação de um caso dentro de um *SRBC* na área do *Direito*.

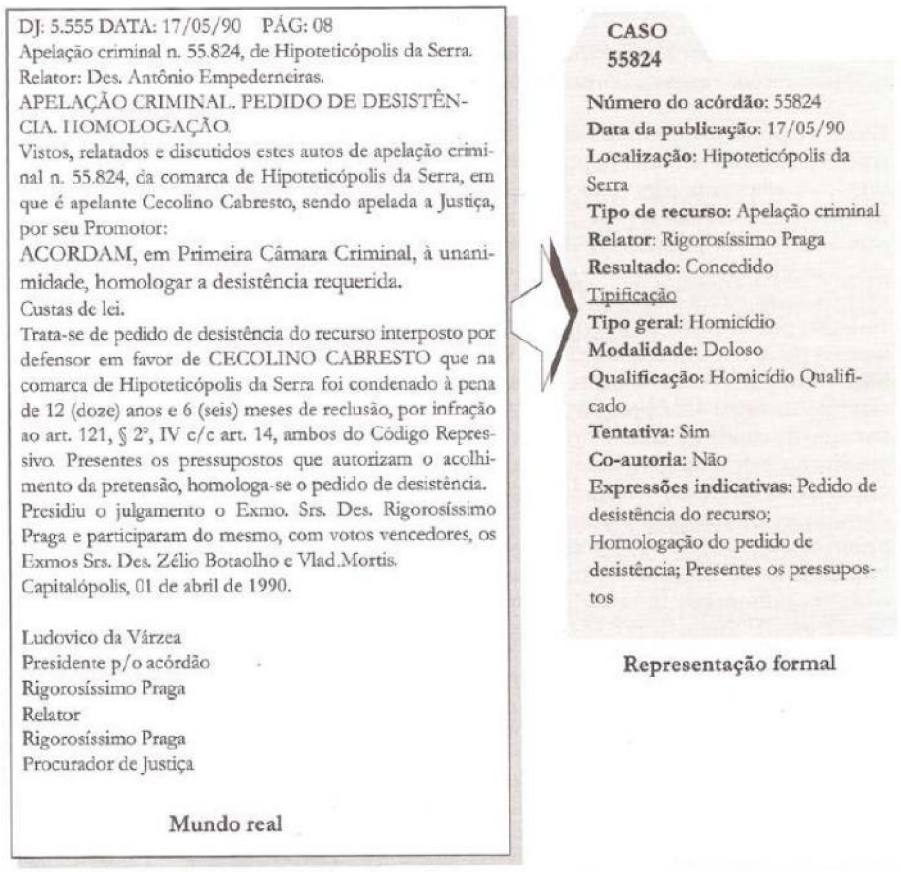

Figura 7 – Exemplo de representação de caso [1].

Como pode ser visto na figura 7, o conteúdo de cada caso varia de acordo com o domínio de aplicação do sistema e o objetivo do raciocínio. Porém, para que o caso atenda às funcionalidades propostas pelo sistema, devem ser levadas em conta a importância da

informação e a facilidade de aquisição da informação. Para tanto, um caso deve ser composto, em geral, pelos seguintes atributos:

- *Descrição do problema*: representa o estado do domínio quando o caso ocorreu. Por exemplo, em Damas seria a representação vetorial de um determinado tabuleiro ou estado do jogo;
- *Solução*: representa a solução adotada para resolver o problema mapeado. A solução pode ser uma ação, um plano ou uma informação útil ao sistema;
- *Avaliação*: também conhecido como *rating*, representa o quanto aquela solução é útil para o problema mapeado.

A *medida de similaridade* – também conhecida como *matching* – entre um novo problema e um dos casos armazenados na *biblioteca de casos* é uma função do tipo $sim(x, y) : (U \times U \rightarrow [0, 1])$, onde x representa o novo problema do domínio, y um caso consultado na *biblioteca*, U o universo de todos os casos armazenados na *biblioteca* do *SRBC* e $[0, 1]$ o *range* de quanto o novo problema x é similar a um caso y da *biblioteca* (0 – sem nenhuma similaridade e 1 – totalmente similar).

Os dois últimos componentes de um *SRBC*, isto é, *adaptação* e *aprendizado*, serão melhores compreendidos na próxima seção.

2.10.1 Ciclo de um sistema RBC

O ciclo de um *SRBC* de acordo com [83] é dividido em quatro etapas principais (os quatro R's). São elas: *recuperar*, *reusar*, *revisar* e *reter*. A figura 8 mostra o ciclo de um *SRBC* que pode ser resumido da seguinte forma: quando um novo problema é apresentadao para um *SRBC*, o sistema tenta recuperar algum caso armazenado na *biblioteca de casos* que mais assemelhe-se ao caso de entrada. Essa é, portanto, a primeira etapa do ciclo, isto é, *recuperação de casos* (1). As próximas duas etapas do ciclo (2 e 3) formam o componente *adaptação* apresentado na seção 2.10. A etapa de *reuso de casos* (2) é responsável por tentar combinar o caso recuperado com o caso de entrada na tentativa de obter um caso solucionado. A etapa de *revisão de casos* (3) verifica se a solução proposta pode ser efetivamente aplicada como solução para o novo problema. A última etapa do ciclo, isto é, *retenção de casos* (4), também conhecida como componente de *aprendizado*, é responsável por verificar se o caso solucionado é uma experiência útil, que seja passível de reutilização no futuro e, então, o armazena na *biblioteca de casos*.

A técnica de *Elicitação Automática de Casos* utilizada neste trabalho é um tipo particular de *SRBC* que adquire, automaticamente, conhecimento, na forma de casos, a partir do zero – isto é, com a *biblioteca de casos* totalmente vazia – e através de interações em tempo real com o ambiente, do tipo *tentativa e erro*, sem depender de nenhum conhecimento pré-codificado do domínio, tais como regras ou *features*, por exemplo. Em [21],

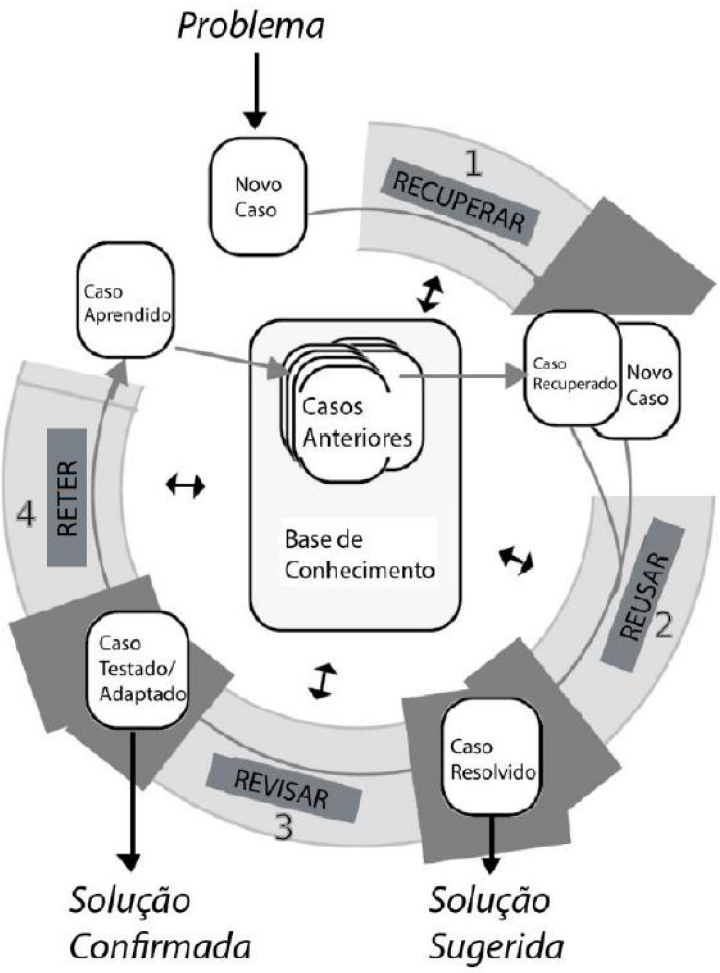

Figura 8 – Ciclo de um sistema *RBC*.

Powell propõe uma versão da técnica *EAC* que adota uma abordagem probabilística para exploração do espaço de busca do jogo de Damas. Nessa abordagem, tanto a geração de novos casos quanto a seleção de casos armazenados na *biblioteca de casos* são conduzidos de forma pseudo-aleatória. Tal versão é a técnica *RBC* adotada pelo presente trabalho para introduzir adaptabilidade de tomada de decisão no sistema *ACE-RL-Checkers*, conforme arquitetura descrita no capítulo 5.

2.11 Mineração de Padrões Sequenciais e Subsequências Binárias Consecutivas

Nos dias atuais, o maior desafio enfrentado por pesquisadores é lidar com um grande volume de dados, armazenados em grandes bancos de dados e *data warehouses*, à procura de padrões consistentes, tais como regras de associação ou sequências temporais, utilizando algoritmos eficientes para descoberta de conhecimento. Dentro desse contexto, o problema de *Mineração de Padrões Sequenciais*, que foi incialmente abordado por [84],

é um caso particular de *Mineração de Dados* e consiste em encontrar, estatisticamente, subsequências frequentes como padrões de conhecimento em um banco de dados de sequências. Em outras palavras, dada uma base de dados de sequências, onde cada sequência é uma lista de transações (ordenadas ou não pelo tempo) e cada transação contém uma lista de itens, o problema relacionado à *MPS* resume-se em encontrar subsequências frequentes, que satisfaçam um suporte mínimo especificado pelo usuário, isto é, localizar aquelas subsequências cuja frequência de ocorrência no conjunto de sequências não seja menor que o suporte mínimo [85].

Um exemplo de banco de dados de sequências é uma base de *registro de jogos* disputados por jogadores de Damas, onde cada partida pode ser representada por uma sequência ordenada de estados de tabuleiro referente ao jogo todo, isto é, $S_1, S_2, S_3, S_4, ..., S_f$, sendo que S_1 e S_n são, respectivamente, estado inicial e final de Damas. A tabela 3 mostra um exemplo simplificado, com bem menos estados de tabuleiro, de um banco de dados de sequências do jogo de Damas que contêm 4 partidas disputadas, onde cada partida ou sequência é identificada por um identificador *ID* (coluna *SID* da tabela 3).

Tabela 3 – Um exemplo de banco de dados de sequências referente a 4 jogos de Damas.

SID	SEQUÊNCIAS
10	<S_1,S_2,S_5,S_8>
20	<S_1,S_3,S_5,S_9,S_{10}>
30	<S_1,S_2,S_4,S_6,S_7>
40	<S_1,S_3,S_5,S_6,S_7>

Considerando o fato que um grande número de possíveis padrões sequenciais podem estar ocultos em um banco de dados, um algoritmo de *MPS* deve ser projetado para [86], [87]:

- ❏ encontrar o conjunto completo de padrões, quando possível, satisfazendo o suporte mínimo (frequência);
- ❏ ser altamente eficiente e escalável, executando uma quantidade mínima e suficiente de varreduras (*scans*) no banco de dados;
- ❏ e, ser capaz de incorporar vários tipos de restrições específicas do usuário.

Nessa linha, muitos esforços foram dedicados ao desenvolvimento de algoritmos de *MPS* eficientes, tais como GSP [84], SPADE [88], CloSpan [89], PrefixSpan [90] e ME-MISP [91]. Recentemente, um novo algoritmo *MPS*, chamado *ErsMining*, foi proposto por [2] com o objetivo de minerar *subsequências binárias consecutivas* – do inglês *Consecutive Binary SubSequences* (CBSS). *CBSS* é um caso particular de *MPS* onde o problema de

minerar *subsequências binárias consecutivas* é descrito a seguir. Seja S um conjunto de sequências de um banco de dados, tal como o exemplo apresentado na tabela 3. Cada *CBSS* consiste de dois estados de tabuleiro que são consecutivos na sequência original e isso pode ser considerado como uma restrição no processo de mineração dos dados. Assim, dado um suporte mínimo, aqui denominado por *minSupport*, o objetivo é encontrar todas subsequências *CBSS* cuja frequência de ocorrência no conjunto S não seja menor que o *minSupport*. Por exemplo, considerando a base S apresentada na tabela 3 e um suporte mínimo $minSupport = 0.5$, ou seja, uma frequência mínima de 2 sequências, de um total de 4 sequências do conjunto S, as seguintes subsequências *CBSS* satisfazem o suporte mínimo estabelecido pelo usuário: $<S_1, S_2>$, $<S_1, S_3>$, $<S_3, S_5>$ e $<S_6, S_7>$. Tais subsequências também podem ser escritas na forma de regras do tipo *se-então*: $S_1 \rightarrow S_2$, $S_1 \rightarrow S_3$, $S_3 \rightarrow S_5$ e $S_6 \rightarrow S_7$.

O algoritmo *ErsMining* proposto por Liang Wang foi testado com sucesso no domínio de Damas Chinesa e mostrou ser mais eficiente do que o tradicional algoritmo *PrefixSpan* [2]. *ErsMining* é o algoritmo *MPS* adotado neste trabalho para minerar *regras de experiência*, a partir de um banco de dados de sequências de movimentos executados por especialistas humanos, de forma a utilizá-las para refinar e acelerar o processo de adaptação do sistema *ACE-RL-Checkers* ao perfil de seu adversário nas fases iniciais de sua interação contra ele. Para mais detalhes sobre tal arquitetura, veja capítulo 6.

Capítulo 3

Trabalhos Correlatos

3.1 Introdução

Devido à vasta disponibilidade de jogadores automáticos fundamentados nas mais diversas técnicas de aprendizagem, este capítulo divide os trabalhos correlacionados em três tópicos principais: agentes com forte supervisão humana, agentes com o mínimo de intervenção humana – aqui classificados como agentes não supervisionados – e agentes que utilizam técnicas para atacar o problema da *modelagem de oponentes*. No grupo dos agentes supervisionados são apresentados os dois principais jogadores para o domínio de Damas: *Chinook* [15] e *Cake* [35]. Esse último é utilizado pelo presente trabalho para realizar alguns testes de performance nas arquiteturas não supervisionadas aqui propostas. No grupo dos agentes não supervisionados e aqueles que adotam técnicas para *modelagem de oponentes*, aplicados tanto para Damas quanto para outros domínios, são apresentados os jogadores automáticos mais relevantes para compreensão dos sistemas *LS-VisionDraughts* e *ACE-RL-Checkers*. Tais agentes são: *NeuroDraughts* [40], *LS-Draughts* [12], [92], *VisionDraughts* [37], *GINA* [93], *CHEBR* [20], [21] e o jogador baseado em *MPS* de Liang Wang [2]. Além disso, ainda no grupo dos jogadores não supervisionados, também são apresentados outros jogadores de Damas disponíveis na literatura que merecem destaque por adotar técnicas de *AM* não supervisionadas.

3.2 Agentes Automáticos de Damas com Aprendizagem Supervisionada

3.2.1 Chinook

O projeto *Chinook* foi iniciado em 1989 com o objetivo de derrotar o campeão de Damas mundial da época – *Marion Tinsley* [15] – e 5 anos mais tarde, em 1994, tornou-se o primeiro campeão de Damas homem-máquina mundial. A arquitetura de *Chinook*

conta com uma coleção de jogadas utilizadas por grandes mestres na fase inicial do jogo, conhecida como *opening books*, e também uma base de dados de final de jogo, conhecida como *endgames*, que contém cerca de 39 trilhões de estados do tabuleiro com valor teórico provado de vitória, empate ou derrota. Essa base de fim de jogo contém todos os possíveis estados de tabuleiros com 10 peças ou menos. Para escolher o melhor movimento de jogo, o jogador utiliza uma versão da técnica de busca *Minimax* distribuída com poda *Alfa-Beta*, TT e aprofundamento iterativo. *Chinook* divide o jogo de Damas em 4 fases, onde cada fase contém 21 *features* que foram ajustadas manualmente para totalizar os 84 parâmetros de sua *função de avaliação*. Esses parâmetros foram ajustados, ao longo de 5 anos, a partir de testes extensivos em jogos contra si mesmo e contra os melhores jogadores humanos, incluindo o *Marion Tinsley* [76].

Em 2007, a equipe do Chinook anunciou que o jogo de Damas estava fracamente resolvido (do inglês *weakly solved*), ou seja, 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 jogador nunca perde, isto é, o jogador pode alcançar o empate contra qualquer oponente jogando tanto com peças pretas quanto com peças brancas [16].

3.2.2 Cake

O *Cake* está entre os jogadores de Damas mais competitivos do mundo. Esse jogador é superior a versão do *Chinook* [15] que venceu o campeonato mundial contra Marion Tinsley [35]. Assim como no *Chinook*, *Cake* também conta com uma base de abertura de jogos (*openning books*) que possui cerca de 2 milhões de movimentos iniciais e uma base de final de jogo (*endgames*) que possui conhecimento perfeito do valor de qualquer movimento para tabuleiros com até 8 peças. Para atuar no jogo entre as fases de inicio e fim de jogo, fases em que o jogador não possui acesso às bases de dados, *Cake* usa o algoritmo de busca *Memory-enhanced Test Driver* (MTD) para escolher o próximo movimento. Tal algoritmo tem capacidade de processar em média 2 milhões de posições por segundo sobre um computador moderno [35].

Cake está disponível através da plataforma *CheckerBoard*, que é a mais completa interface livre para agentes jogadores automáticos de Damas [36]. Essa plataforma suporta arquivos *Portable Draughts Notation* (PDN), que é o formato padrão para arquivos de jogos de Damas. A *CheckerBoard* disponibiliza também base de dados contendo milhares de jogos de competições entre os grandes jogadores de Damas, incluindo jogos entre *Marion Tinsley* e *Chinook*.

3.3 Agentes Automáticos com Aprendizagem Não Supervisionada

3.3.1 Agentes Aplicados ao Domínio de Damas

3.3.1.1 NeuroDraughts

NeuroDraughts é um agente automático que aprende a jogar Damas sem nenhuma supervisão humana [40] e cuja arquitetura é a base de desenvolvimento do presente trabalho. Sua arquitetura é apresentada na figura 9. O *módulo da RNA* consiste em uma *MLP* que é a própria *função de avaliação* que define o agente. O papel da *MLP* é estimar o quanto o estado de tabuleiro corrente do jogo – representado na entrada da *MLP* através do mapeamento *NET-FEATUREMAP* com 12 *features* (conforme explicado na seção 2.6) – é favorável para o agente. Tal estimativa é conhecida por *predição*. O *módulo de busca Minimax* é responsável por retornar qual a melhor ação a ser executada pelo agente, a partir de um determinado estado corrente, utilizando as *predições* da *MLP* para os nós folha da árvore do jogo expandida para uma determinada profundidade. *NeuroDraughts* também utiliza o *módulo de aprendizagem por reforço TD(λ)* aliado à estratégia de treinamento *self-play* com clonagem (um tipo de treinamento onde o agente enfrenta uma cópia de si próprio – mais detalhes são apresentados na seção 4.4.2) como ferramentas para atualizar os pesos sinápticos da *MLP*.

O processo de interação entre os 3 módulos da figura 9 é resumido a seguir. Sempre que o agente está em processo de treinamento e precisa escolher um novo movimento, aqui denominado por a_{t+1}, pois $t+1$ representa um tempo futuro ao estado ou tempo corrente t, os passos #1 a #6 da figura 9 são acionados. O estado de tabuleiro corrente, aqui denominado por S_t, cuja predição P_t foi calculada no ciclo anterior relacionado à escolha do último movimento, é apresentado ao *Módulo RNA* #1 (passo 1 na figura). Esse módulo chama o *Módulo de Busca Minimax*, #2, que monta uma árvore de busca do jogo cuja raiz é o estado S_t e retorna o melhor movimento a_{t+1} a ser executado em S_t, #3. O agente então executa esse movimento, #4. O novo estado de tabuleiro S_{t+1} é apresentado para a *MLP* avaliá-lo gerando uma predição P_{t+1}, #5. As predições P_{t+1} referente ao estado S_{t+1} (novo estado) e P_t referente ao estado S_t (antigo estado corrente cuja predição foi calculada no ciclo anterior) são usadas pelo *Módulo de Aprendizagem TD* como parâmetros de entrada do método TD(λ) para ajustar os pesos da rede *MLP*, #6. Durante os jogos não treino (ou seja, após a etapa de treinamento), o processo é análogo ao ciclo de treinamento apresentado na figura 9, exceto nos seguintes aspectos: o *Módulo de Aprendizagem TD* e os passos #5 e #6 devem ser desconsiderados.

É importante destacar que apesar da arquitetura do agente *NeuroDraughts* ser uma proposta interessante de agente automático que aprende por reforço, tal jogador apresenta três pontos fracos principais: a escolha manual de 12 *features* dentre aquelas propostas

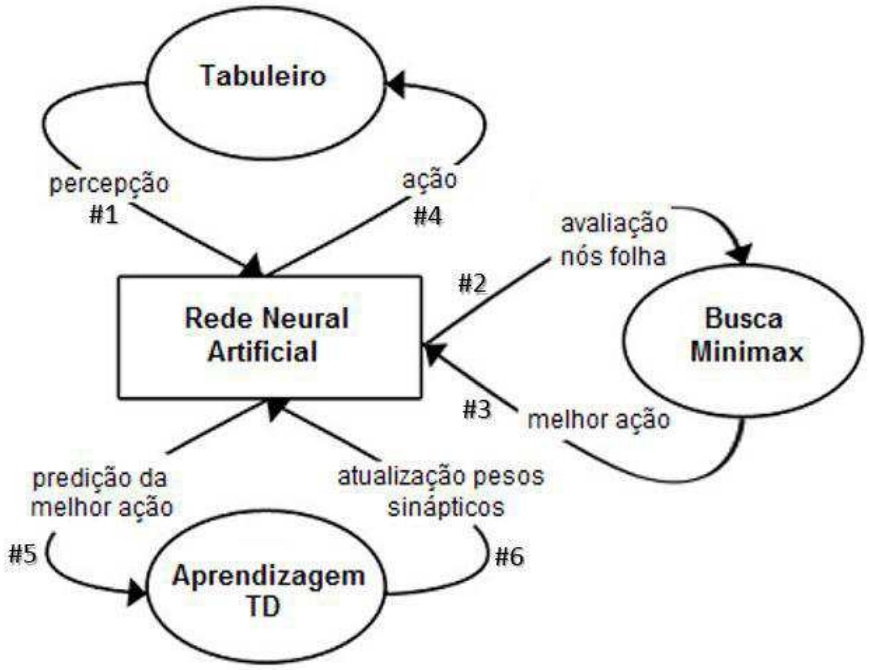

Figura 9 – Arquitetura geral do *NeuroDraughts*.

por Samuel em [13] e [14]; o baixo desempenho de seu algoritmo de busca *Minimax*; e a ocorrência frequente de *loops* de final de jogo – situações em que o agente, apesar de estar em uma situação de vantagem no jogo em relação ao seu oponente, não consegue finalizá-lo e consequentemente, executa, repetidamente e infinitamente, sempre o mesmo movimento. Nesses casos, é comum que os projetistas contem indevidamente esses *loops* como empate (mais detalhes veja [92]).

3.3.1.2 LS-Draughts

Visando atacar o primeiro ponto fraco identificado na arquitetura do jogador *Neuro-Draughts*, isto é, a escolha manual de suas *features* (conforme citado na seção 3.3.1.1), Neto e Julia propuseram o agente *LS-Draughts*: um sistema que gera, automaticamente, por meio da técnica dos *AGs*, um conjunto de *features* mínimas necessárias e essenciais para o jogo de Damas, de forma a otimizar o treino de um agente automático [12] e [92].

A figura 10 mostra a arquitetura geral do *LS-Draughts* que consiste basicamente em um sistema de aprendizagem evolutivo no qual 50 indivíduos evoluem ao longo de 30 gerações. Cada indivíduo ou cromossomo representa um subconjunto de 15 *features* (12 *features* de Lynch mais 3 *features* selecionadas do jogador de Samuel) que é acoplado a uma rede *MLP* cujos pesos são atualizados pelos métodos TD(λ) utilizando a estratégia de treinamento por *self-play* com clonagem. O mapeamento *NET-FEATUREMAP* é utilizado para representar os estados do tabuleiro do jogo na entrada da *MLP*. Tais representações baseiam-se em conjuntos de *features* gerados, automaticamente, pelo *AG*. O módulo de busca utilizado é o mesmo do *NeuroDraughts*, isto é, o tradicional algoritmo

de busca *Minimax*.

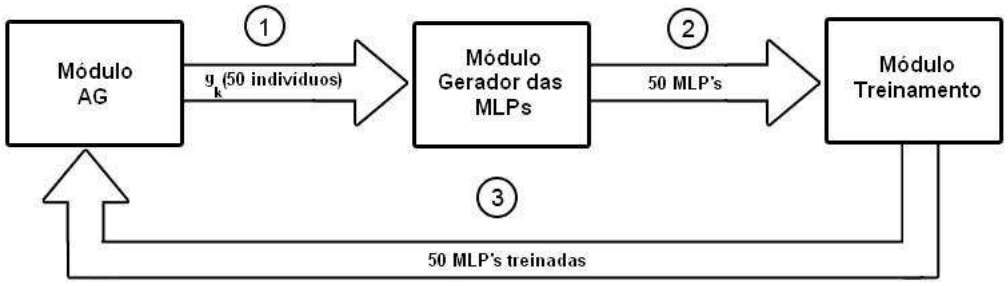

Figura 10 – Arquitetura do LS-Draughts

Os resultados obtidos por [12] mostraram que a inclusão do módulo do *AG* possibilitou ao *LS-Draughts* escolher um conjunto mínimo de atributos que melhor caracterizam o domínio de Damas e que sirvam como um meio pelo qual a *função de avaliação* adquirirá novos conhecimentos, o que é uma questão fundamental para acelerar a aprendizagem e obter novos agentes com alto nível de desempenho. Assim, com o processo de escolha de *features* automatizado, *LS-Draughts* conseguiu vencer o jogador *NeuroDraughts* utilizando apenas 7 *features* (contra 12 *features* utilizadas pelo jogador de Lynch): o melhor indivíduo da geração 25 derrotou *NeuroDraughts* com 2 vitórias e 5 empates, em um torneio de 7 jogos [12].

Apesar do fato de *LS-Draughts* ter melhorado o desempenho geral do *NeuroDraughts* usando *AG*, ele não conseguiu atenuar significativamente o problema do *loop* de final de jogo: reduziu apenas 5% a taxa de ocorrência de *loops* de final de jogo gerada pelo *NeuroDraughts* [12]. É por isso que em [92], os autores atacaram esse problema inserindo dentro da arquitetura do *LS-Draughts* as bases de final de jogo do *Chinook*. Nessa versão estendida os indivíduos foram treinados utilizando bases de final de jogo para antecipar o resultado do jogo (vitória, derrota ou empate) para os estados de tabuleiro com, no máximo, 8 peças. Muitos torneios utilizando a versão estendida do *LS-Draughts* confirmaram a contribuição da base de final de jogo para atenuar o problema do *loop*: redução de 83% e de 77% na taxa de ocorrência de *loops* de final de jogo gerada pelo *NeuroDraughts* e pela versão original do *LS-Draughts*, respectivamente.

3.3.1.3 VisionDraughts

Visando atacar o segundo ponto fraco identificado na arquitetura do jogador *NeuroDraughts*, isto é, o baixo desempenho de seu algoritmo de busca *Minimax* (conforme citado na seção 3.3.1.1), Caixeta e Julia propuseram o agente *VisionDraughts*: jogador automático que substitui o módulo de busca *Minimax* pela busca com poda *Alfa-Beta*, estratégia de aprofundamento iterativo e TT. O objetivo com tal proposta é acelerar o processo de busca e melhorar a visão *look-ahead* do agente. Entretanto, por outro lado, *VisionDraughts* faz uso das mesmas 12 *features* escolhidas manualmente pelo projetista

do *NeuroDraughts* [37].

Com a arquitetura do *VisionDraughts*, foi possível obter um jogador muito melhor que o *NeuroDraughts*, mantendo o mesmo tempo de treinamento e aumentando consideravelmente a profundidade da busca. Resultados apresentados em [37] comprovam que o *VisionDraughts* é, pelo menos, 50% mais eficiente com tempo de busca 95% inferior quando comparado com o agente *NeuroDraughts*.

Com o objetivo de também observar a contribuição da base de final de jogos do *Chinook* na tratativa do *loop* de final de jogo, os autores inseriram-na na arquitetura do *VisionDraughts*, obtendo os mesmos resultados significativos produzidos pela versão estendida do *LS-Draughts*, isto é, uma redução de 83% na taxa de ocorrência de *loops* de final de jogo [37].

Baseado nos sucessos das arquiteturas dos jogadores *LS-Draughts* e *VisionDraughts* que a construção do sistema *LS-VisionDraughts* é proposto como primeiro objetivo deste trabalho: obter um sistema evolutivo melhor do que ambos e com a capacidade de operar com eficiência em um tempo razoável. Maiores detalhes sobre a arquitetura do sistema *LS-VisionDraughts* são apresentados no capítulo 4.

3.3.1.4 Anaconda e a Computação Evolutiva

A aplicação da *Computação Evolutiva* em jogos tem-se mostrado bastante eficiente na obtenção de bons agentes jogadores, tornando-se, assim, um paradigma alternativo ao processo de treinamento convencional. Nesse tipo de treinamento, as melhores soluções tendem a evoluir com o passar das gerações e as piores soluções tendem a desaparecer. Nesse sentido, Fogel [17], [94] utilizou um processo evolutivo para implementar um agente automático de Damas que aprende a jogar sem utilizar qualquer tipo de perícia humana na forma de *features* específicas do domínio do jogo. O jogador de Fogel utiliza um mapeamento espacial (combinações das áreas do tabuleiro do jogo) para representar o tabuleiro de Damas na entrada de uma rede *MLP*. O melhor jogador de Fogel, chamado *Anaconda*, foi resultado da evolução de 30 redes *MLPs* ao longo de 840 gerações. Cada geração teve em torno de 150 jogos de treinamento, sendo 5 jogos para cada um dos 30 indivíduos da população. Assim, foram necessários 126.000 jogos de treinamento e 6 meses de execução para obtê-lo. Em um torneio de 10 jogos contra uma versão de baixo nível do *Chinook*, versão que adota o algoritmo de busca com profundidade 5, o Anaconda obteve 2 vitórias, 4 empates e 4 derrotas, o que permitiu classificá-lo como especialista (do inglês *expert*).

Atualmente, escolher um método evolutivo ou *AR* como TD(λ) para treinar uma rede neural pode ser uma tarefa árdua. Graças ao sucesso de ambas técnicas em alguns domínios específicos como Gamão [95], [96] e Xadrez [97], a conclusão é que ainda existe muito a ser explorado nessa área do conhecimento humano [12]. Por outro lado, Paul Darwen demonstra em [98] a vantagem de utilizar TD(λ) no treinamento de *MLPs* por causa da rapidez com que a rede aprende um comportamento não linear sobre um determinado

problema. Darwen demonstra essa questão ao discutir o porquê da evolução conseguir bater, para uma arquitetura de rede linear, isto é, perceptron simples, a aprendizagem por TD(λ) no jogo do Gamão, mas não conseguir o mesmo feito para uma arquitetura de rede não linear, isto é, rede neural com camada oculta. O autor mostra que, se são necessários bilhões de jogos para que uma arquitetura não-linear treinada por um método evolutivo consiga bater uma outra arquitetura não-linear treinada pelo método TD(λ), a qual, por sua vez, requer apenas cerca de 100.000 jogos para aprender, então muitos dos bilhões de jogos do método evolutivo não estarão, de fato, contribuindo para a aprendizagem.

Esse fato demonstrado por Paul Darwen parece também ser aplicado ao domínio de Damas, quando pretende-se treinar uma *MLP* por meio de um algoritmo evolutivo. Por exemplo, o jogador *Anaconda* precisou de 126.000 jogos de treinamento para apresentar o mesmo nível de desempenho do *Chinook*, enquanto que o jogador de Damas de Schaeffer, treinado pelo método TD(λ) [19], precisou de apenas 10.000 jogos de treinamento para obter o mesmo resultado. Outro exemplo é o agente *NeuroDraughts* proposto por [6], [40], o qual foi obtido depois de apenas 2.000 jogos de treinamento com o método TD(λ) e utilizando um conjunto de *features* selecionadas manualmente para representar o tabuleiro de Damas na entrada da *MLP* [12].

Baseado nesses fatos, o presente trabalho propõe para o sistema *LS-VisionDraughts* a manutenção do método TD(λ) como técnica de aprendizagem e ajuste dos pesos sinápticos da *MLP* e adota o *AG* como técnica de seleção automática das *features* do mapeamento *NET-FEATUREMAP*.

3.3.1.5 Outros Agentes Automáticos de Damas Não Supervisionados

Dentre os agentes automáticos de Damas não supervisionados propostos recentemente na literatura, alguns podem ser destacados:

- O jogador de Cheheltani e Ebadzadeh, [22], corresponde a um agente imunológico difuso que usa células de memória (do inglês *memory cells*) para representar ações do jogo de Damas. Essas células ajudam um sistema de inferência difuso de Mamdani (do inglês *Mamdani Fuzzy Inference Engine* – FIS) a decidir qual é o melhor movimento a ser executado com base nas análises dos estados anteriores e posteriores ao tabuleiro corrente do jogo. Em um torneio de 50 jogos contra o jogador de Fogel publicado em [18], o agente imunológico obteve 66% de vitórias contra 10% de vitórias do oponente [22];

- O jogador de Al-Khateeb e Kendall, [23], [25], corresponde a uma evolução do jogador do Fogel proposto em [18], o qual acrescenta um mecanismo de *aprendizagem social e individual* dentro da fase de ajuste dos pesos da rede *MLP* evolutiva com o objetivo de melhorar seu processo de aprendizagem. Resultados obtidos em torneio

mostraram que o novo sistema evolutivo proposto é superior à versão do jogador de Fogel publicada em [18];

- Em [29] os autores propõem a inclusão da poda *alfa-beta* e aprofundamento iterativo no método de busca em árvore chamado *NegaMax*, uma variante do tradicional algoritmo *Minimax* em que apenas as operações de maximização são consideradas em todos os níveis da árvore. Nesse caso, nos níveis de minimização da árvore, os valores retornados pelos filhos são transformados em valores negativos para que tal operação possa ser executada. Para avaliar os nós folha da árvore, os autores propuseram uma função de avaliação que define os valores (*scores*) para cada estado do tabuleiro baseado nos seguintes parâmetros: localização (o tabuleiro é dividido em 6 regiões), tipo de peça (simples ou dama), possibilidade de captura de peças adversárias e *layout* de posicionamento regional das peças. Tal algoritmo de busca foi aplicado com sucesso em um tabuleiro de Damas 10×10 [29]. Um outro trabalho que também explora uma variante do algoritmo *NegaMax* no domínio de Damas é o agente *MPACA* proposto por [28]. Tal agente é composto por um módulo de processamento de imagem para detectar os movimentos humanos (contra quem ele joga), um módulo robótica para executar os movimentos do agente sobre um determinado tabuleiro e um módulo de *IA* que é responsável por definir qual a melhor ação a ser executada. Tal módulo é composto por uma versão distribuída do algoritmo *NegaMax* com poda *alfa-beta* e utilizando uma base de dados paralela que armazena e recupera valores de tabuleiros avaliados por uma função de avaliação qualquer.

3.3.2 Agentes Aplicados a Outros Domínios

3.3.2.1 GINA

Em [93] De Jong descreve que a aplicação adequada de algoritmos *EBL*, tais como um *SRBC* por exemplo, em combinação com um *solucionador de problemas* (agente) estático pode conduzir ao desenvolvimento de um sistema híbrido muito melhor que esse último e com grande capacidade de recuperar e aplicar rapidamente as ações executadas no passado e que estão armazenadas em uma *base de conhecimento*. Nessa abordagem híbrida proposta por De Jong, as ações são inicialmente sugeridas pelo agente e vão sendo armazenadas na *base de conhecimento*. Ao longo do tempo, o sistema ganha experiência e então, passa a aplicar somente aquelas ações que provaram produzir melhores resultados para o agente. Para demonstrar a eficiência de tal abordagem, De Jong constrói um sistema chamado *GINA* que é um agente jogador de Othello que utiliza *RBC* como técnica *EBL* e um agente estático com *look-ahead Minimax*, isto é, que faz uso da busca em árvore *Minimax*, como suporte para a *base de experiência ou conhecimento* do *RBC*. Assim, toda vez que *GINA* está jogando e encontra um cenário de jogo (tabuleiro) que não existe em

sua *base de conhecimento*, ele primeiro aciona seu agente estático jogador de Othello (*solucionador de problemas*) para escolher o melhor movimento a ser executado e então, armazena o movimento sugerido em sua *base de conhecimento*. Ao fim de cada jogo, um algoritmo *Minimax* é usado para repartir o crédito de cada movimento executado durante o jogo e atualizar o *rating* (avaliação) de cada caso na *base de conhecimento* do sistema híbrido. GINA foi testado com sucesso contra diversas versões estáticas de agentes jogadores de Othello disponíveis na literatura [93]. Essa mesma versão de *GINA* foi reproduzida para o jogo de Damas e testada contra a versão probabilística da técnica *EAC* proposta em [21], conforme será apresentado na seção 3.4.1.

3.3.2.2 Agente baseado em MPS de Liang Wang

Recentemente um novo algoritmo *MPS*, chamado *ErsMining*, foi proposto por Liang Wang com o objetivo de minerar *padrões de experiência* a partir de *log de jogos* gerados por uma plataforma cliente que permite dois jogadores (humanos e/ou automáticos) disputarem jogos de Damas Chinesa *online* [2]. A figura 11 mostra um esboço da arquitetura geral de tal plataforma que é conectada a um sistema *MPS* servidor através de trocas de arquivos XMLs. Nessa arquitetura, o lado do cliente, que corresponde à plataforma que controla o jogo de Damas Chinesa, é responsável por gerar arquivos XMLs à medida que cada jogo *online* é finalizado. Já o lado do servidor, que corresponde ao sistema *MPS*, é responsável por processar a base de *log de jogos* através do algoritmo *ErsMining* com o objetivo de minerar *padrões de experiência*, que depois são salvos em arquivos XML, para que possam ser utilizados por um agente automático que joga na plataforma cliente. Tal agente foi projetado por [2] para aprender a jogar Damas Chinesa utilizando apenas o conhecimento extraído a partir de sequências de movimentos provenientes de jogos *online* disputados entre jogadores humanos ou entre o agente e um adversário humano. Com essa arquitetura, Liang Wang justifica que a aprendizagem do agente é sempre contínua e crescente à medida que a base de *log de jogos* é enriquecida [2].

Em relação ao sistema *MPS* proposto por [2], ele é composto por 3 fases:

1. *Extração de dados:* fase responsável por ler os arquivos XMLs, correspondentes aos jogos armazenados pela plataforma cliente, e gerar um banco de dados sequenciais;

2. *Mineração de dados:* fase que adota um conjunto de algoritmos baseados em árvore de sequência, *SequenceTreeCreation*, *BranchShifting*, *CBSSFinding* e *ReliabilitySelection*, cujos nomes foram unificados para *ErsMining* com o objetivo de minerar três tipos de *padrões de experiência*:

   - *Regras de Experiência* ou *Experiential Rules*: esse padrão é similar a uma regra do tipo *se-então*, isto é, se um determinado tabuleiro de jogo S_a aparece, então o agente deverá alcançar o estado de tabuleiro S_b através da execução de um

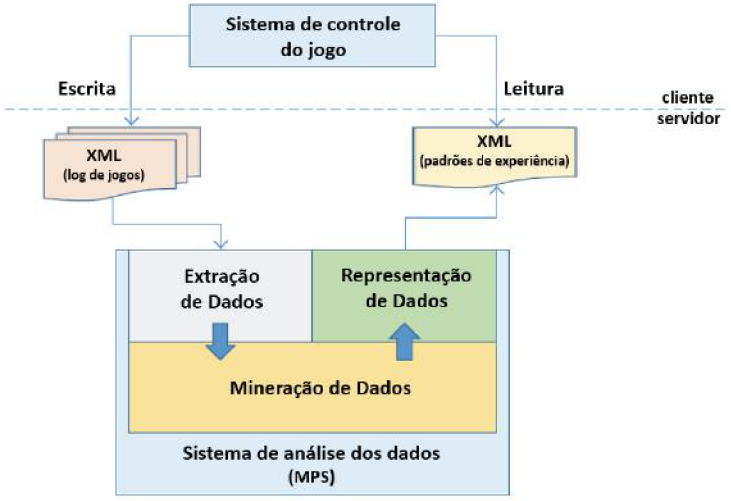

Figura 11 – Arquitetura geral da plataforma jogadora de Damas Chinesa proposta em [2].

determinado movimento sobre o tabuleiro. Assim, uma *regra de experiência* pode ser representada como $S_a \rightarrow S_b$, onde S_a corresponde ao estado corrente e S_b corresponde ao próximo estado a ser alcançado pelo agente;

- *Estados-Chave* ou *Key States*: esse padrão corresponde a um grupo particular de estados de tabuleiro que aparecem com frequência nos jogos de vitória e, portanto, indicam uma boa situação de jogo para o agente. Assim, a partir de um algoritmo é possível calcular estados de tabuleiros que estejam mais próximos aos *estados-chave* e então, definir qual o melhor movimento a ser executado;
- *Casa de Tabuleiro em Uso* ou *Checker Usage*: esse padrão armazena a quantidade de vezes que uma determinada casa do tabuleiro de Damas Chinesa é utilizada como movimento de destino de um jogador. Neste sentido, todas as casas do tabuleiro possuem um valor de *checker usage* associado.

3. *Representação de dados:* essa fase é responsável por armazenar, em arquivos XMLs, os *padrões de experiência* minerados na etapa 2 para que possam ser utilizados pelo agente automático de Liang Wang com o objetivo de auxiliá-lo na escolha do melhor movimento a ser executado em um determinado tabuleiro do jogo.

O processo de tomada de decisão do agente de Liang Wang é resumido a seguir. No decorrer de um jogo, o agente primeiro tenta utilizar alguma *regra de experiência* aplicável para um determinado estado de tabuleiro S_i. Se sim, o agente apenas executa o movimento que leva para o estado S_{i+1} proveniente da regra aplicada. Caso contrário, quando não há *regras de experiência* disponíveis para o tabuleiro S_i, então o agente tenta executar um movimento que leva a um estado S_{i+1} que mais aproxime aos *estados-chave* através do menor valor de dissimilaridade. Se, entretanto, a dissimilaridade entre o *estado-chave*

mais próximo e o estado corrente S_i é maior do que um determinado limiar, então o agente executa um algoritmo proposto por [2], chamado *TentativeMove*, que é baseado no tradicional algoritmo de busca *alpha-beta* e faz uso do terceiro padrão de experiência minerado, isto é, *casa de tabuleiro em uso* (para mais detalhes sobre o algoritmo *TentativeMove*, veja [2]).

Ao longo dos testes experimentais realizados por Liang Wang, um total de 1.350 *log de jogos* foram armazenados pela plataforma cliente, 528 *regras de experiência* e 575 *estados-chave* foram minerados utilizando o algoritmo proposto *ErsMining*. Tal algoritmo foi comparado com o tradicional algoritmo *PrefixSpan* e demonstrou ser mais eficiente que esse último. Os resultados também demonstraram que a nova abordagem de *MPS* proposta por Liang Wang é efetiva e eficiente, uma vez que permite que agentes automáticos possam aprender, progressivamente, a partir de experiência humana lida de uma base crescente de *log de jogos*. Além disso, os autores também citam que a nova abordagem *MPS* proposta é fácil de ser aplicada para outros tipos de jogos [2].

Motivado pelos resultados obtidos em [2], este trabalho propõe a inserção de um novo módulo na arquitetura do sistema *ACE-RL-Checkers* composto por uma base de *regras de experiência*, isto é, o primeiro tipo de *padrão de experiência* proposto por Liang Wang em [2], minerada a partir de *log de jogos* contendo sequências de movimentos executados por especialistas de Damas. O objetivo, no caso, é utilizar tais *regras de experiência* nas fases iniciais do jogo de Damas de forma a refinar e acelerar o processo de adaptação dinâmica do agente ao perfil de seu adversário. Maiores detalhes sobre a arquitetura estendida do sistema *ACE-RL-Checkers* são apresentados no capítulo 6.

3.4 Agentes Automáticos que Atacam o *Problema da Modelagem de Oponentes*

Em [7], os autores definem uma taxonomia para *modelagem de oponentes* que consiste, basicamente, em categorizar diversos tipos de implementação de modelos de agentes automáticos. A tabela 4, extraída de [7], resume os principais componentes para modelagem de um oponente, que vão desde a descrição abstrata do estado corrente de um jogador em um determinado momento do jogo, isto é, o que um estado do oponente representa: *satisfação*, *conhecimento*, *posição* e/ou *estratégia*, até o tipo de implementação da própria modelagem. Nesse caso, a modelagem pode ser *explícita*, que ocorre quando a especificação dos atributos do oponente existe separadamente do processo de tomada de decisão, ou *implícita*, que é quando os atributos são geralmente incorporados e diluídos em diferentes partes do código, o que torna a tarefa de identificar e descrever estes atributos mais difícil.

Por fim, Marlos cita que a representação interna de um modelo do oponente depende do tipo de conhecimento do qual ele deve ser dotado e da tarefa que executará. Para tanto, os autores listam uma série de técnicas aplicáveis, como por exemplo, função de

Tabela 4 – Taxonomia para *modelagem de oponentes* em jogos extraída de [7].

Descrição	Categorias	Metas	Aplicações	Métodos	Implementação
Conhecimento	Rastreamento Online	Colaborativa	Busca e Planejamento	Modelagem de Ação	Explícita
Posição	Estratégia Online	Oposta	Orientação	Modelagem de Preferência	Implícita
Estratégia	Revisão Off-line	Criação de Estórias	Treinamento	Modelagem de Posição	-
Satisfação	-	-	Substituição	Modelagem de Conhecimento	-

avaliação, redes neurais, modelos baseados em regras, máquinas de estado finito, modelos probabilísticos, modelos baseado em casos, algoritmos genéticos, programação genética, busca Monte Carlo e outras [7].

A próxima seção apresenta o agente não supervisionado *CHEBR*, jogador que utiliza uma técnica de *RBC* para modelar implicitamente o perfil de jogo do oponente através da exploração pseudo-aleatória no espaço de busca do jogo de Damas. Tal arquitetura é inspiração para a construção do sistema *ACE-RL-Checkers* proposto neste trabalho. Na sequência, outros agentes publicados recentemente na literatura que atacam o problema da *modelagem de oponentes* são apresentados.

3.4.1 CHEBR – CHeckers case-Based Reasoner

CHEBR [20], [21] é um sistema automático que aprende a jogar Damas do zero, isto é, sem qualquer tipo de conhecimento do jogo, utilizando apenas a técnica de *AM* não supervisionada *Elicitação Automática de Casos*. Em [20] Powell testa dois tipos de algoritmos de *matching* (também conhecido como *medida de similaridade*) para recuperar casos da *biblioteca EAC* e uma versão da técnica *EAC* que corresponde a um *SRBC* não probabilístico. O primeiro algoritmo de *matching*, chamado *CHEBR-Exact*, requer que casos recuperados da *biblioteca EAC* correspondam exatamente ao tabuleiro corrente do jogo (novo problema). O segundo algoritmo de *matching*, chamado *CHEBR-Region*, usa um esquema de *matching regional*, isto é, utiliza regiões específicas do tabuleiro de Damas, para recuperar casos similares da *biblioteca EAC*. Uma vez recuperado os casos aplicáveis ao tabuleiro do jogo corrente, a versão não probabilística da técnica *EAC* seleciona o caso com o melhor desempenho – tupla (tabuleiro,ação) que produziu melhor resultado para o agente até aquele momento, isto é, caso com o maior valor de *rating* [20]. Caso não exista nenhum caso armazenado na *biblioteca*, isto é, o tabuleiro corrente do jogo é um novo problema não mapeado pelo agente, então uma nova ação aleatória é executada. Testes realizados em torneios com 2.000 jogos mostraram que a versão *CHEBR-Exact* é superior à versão *CHEBR-Region* [20].

Em [21], Powell adota o algoritmo de *matching CHEBR-Exact* e propõe uma variação da técnica *EAC* adotada em [20] na qual introduz uma abordagem probabilística para seleção pseudo-aleatória de casos. Em tal abordagem quanto melhor é um caso, maior é a probabilidade do mesmo ser selecionado dentre todos os casos aplicáveis a um determinado tabuleiro corrente do jogo. A versão da técnica *EAC* proposta em [21] difere daquela proposta em [20] por utilizar um *SRBC* probabilístico que possui a habilidade de explorar

aleatoriamente o espaço de busca, uma vez que estimula a exploração de novos casos -– já que nem sempre o melhor caso é selecionado. A versão probabilística de *CHEBR* [21] foi testada contra 3 versões diferentes de agentes de Damas: um agente estático com *look-ahead* de profundidade 4 que utiliza o algoritmo de busca *Minimax* com poda *Alfa-Beta*; um agente que é uma versão para Damas similar ao sistema GINA implementado por De Jong em [93] (detalhes desse sistema são apresentados na seção 3.3.2.1); por fim, um agente que é uma versão de não exploração do *CHEBR* (do inglês *Non-Explore CHEBR*) é investigado. Em tal versão, a habilidade de exploração aleatória, isto é, seleção e geração aleatória de casos da *biblioteca*, foram removidos e substituídos por um *look-ahead* de profundidade 4. Em um torneio com 500 jogos, a versão probabilística do agente *CHEBR* superou todas as 3 versões dos jogadores de Damas investigadas [21].

É importante destacar que apesar da arquitetura híbrida proposta pelo sistema *GINA*, descrita na seção 3.3.2.1, ter conseguido melhorar suas habilidades em relação a vários agentes estáticos de Othello disponíveis na literatura [93], a versão do *SRBC* adotada por De Jong é não probabilística e tende a tornar-se estática ao longo do tempo. Isso acontece porque o modulo de seleção de ação adotado por *GINA* baseia-se apenas nas sugestões de movimento do agente estático em conjunto com os resultados das experiências experimentadas pelo sistema com a execução de tais ações. Em outras palavras, tal arquitetura não possui a habilidade de experimentar novas ações, além daquelas sugeridas pelo agente estático, e consequentemente, nem de explorar novas regiões no espaço de busca. Esse problema foi identificado por Powell em [21] ao reproduzir o sistema *GINA* para o domínio de Damas com o propósito de comparar o desempenho da técnica *EAC* (um *SRBC* probalilístico) contra a técnica proposta por De Jong (um *SRBC* não probabilístico). A superioridade de *CHEBR*, que não conta com o conhecimento de um agente estático, em relação ao sistema *GINA* é justificada em [21] pelo poder da exploração aleatória da técnica *EAC* e pela capacidade de tal técnica modelar implicitamente o perfil de jogo do oponente ao longo dos jogos.

Motivado por esses fatos que o presente trabalho propõe a construção do sistema híbrido *ACE-RL-Checkers*: uma nova abordagem de *AM* que unifica as habilidades das técnicas *EAC* e *AR* com o objetivo de aprimorar a exploração aleatória realizada pelos agentes baseados apenas em *EAC* e introduzir adaptabilidade de tomada de decisão nos agentes baseados apenas em *AR*. Maiores detalhes sobre a arquitetura do sistema *ACE-RL-Checkers* são apresentados no capítulo 5.

3.4.2 Outros Agentes Automáticos que Tratam o *Problema da Modelagem de Oponentes*

Dentre os agentes automáticos, publicados recentemente na literatura e que atacam o problema da *modelagem de oponentes*, dois trabalhos podem ser destacados:

- Frankland e Pillay propuseram em [30] uma arquitetura híbrida jogadora de Damas composta pelas técnicas *Programação Genética* (PG) e *AR* para evoluir estratégias do jogo de Damas, em tempo real, baseadas em heurísticas. Tais heurísticas dividem o tabuleiro de Damas em áreas estratégicas com base em informações importantes do jogo, tais como, regiões propícias para ataque e defesa. Assim, cada indivíduo da *PG* corresponde a uma estratégia diferente que, através de um conjunto de heurísticas, define qual o movimento a ser executado sobre um determinado tabuleiro. O processo evolutivo do sistema proposto por [30] é resumido a seguir. Toda vez que o sistema deve executar um movimento sobre um determinado tabuleiro do jogo, uma execução separada de um ciclo inteiro da *PG* é realizada com o objetivo de obter o melhor indivíduo (estratégia), também chamado pelos autores como *jogador alfa*, o qual é responsável por definir a ação a ser executada nesse tabuleiro. Este processo repete até o fim do jogo e usa *AR* para melhorar as estratégias evoluídas. Os resultados obtidos em torneio confirmam que a nova abordagem proposta é bastante promissora [30];

- Em [99], os autores propõem a construção do jogador *DeepGo* que corresponde a uma *rede neural convolucional profunda* de 12 camadas que aprende a predizer movimentos de especialistas humanos através da técnica de aprendizagem supervisionada *descida gradiente estocástica assíncrona*. O treinamento é realizado sobre uma base de jogos de especialistas de Go de diferentes níveis de habilidade e para diferenciar tais habilidades, os autores incluem na arquitetura da rede convolucional uma entrada global adicional que informa o *rank* (codificado de 1 a 9) de habilidade ou força do jogador que é lido a base de seus jogos. Um trabalho similar também é proposto em [64] com o agente *AlphaGo*. Tal agente conseguiu superar o campeão de Go europeu utilizando *redes neurais convolucionais profundas* treinadas através de uma abordagem híbrida que combina *aprendizagem por reforço* com *aprendizagem supervisionada*.

3.5 Resumo dos Trabalhos Correlatos

A tabela 5 resume os trabalhos correlatos apresentados neste capítulo.

Tabela 5 – Resumo dos trabalhos correlatos.

Categoria	**A**gente Jogador	**R**eferência Bibliográfica
Agentes de Damas Supervisionados	*Chinook*	[15]
	Cake	[35]
Agentes de Damas Não Supervisionados	*NeuroDraughts*	[40]
	LS-Draughts	[12]
	VisionDraughts	[37]
	Anaconda	[17]
	O jogador de Cheheltani e Ebadzadeh	[22]
	O jogador de Al-Khateeb e Kendall	[23], [25]
	O jogador de Zhao	[29]
	MPACA	[28]
Agentes Não Supervisionados Aplicados a Outros Domínios	*GINA* (Othello)	[93]
	O jogador de Liang Wang (Damas Chinesa)	[2]
Agentes Automáticos que atacam o problema da *Modelagem de Oponentes*	*CHEBR* (Damas)	[20], [21]
	O jogador de Frankland e Pillay (Damas)	[30]
	DeepGo e *AlphaGo* (Go)	[99], [64]

Capítulo 4

LS-VisionDraughts

Este capítulo apresenta o sistema evolutivo não supervisionado *LS-VisionDraughts* referente ao primeiro objetivo traçado pelo presente trabalho e que foi descrito na seção 1.3.1. Os tópicos a seguir resumem as principais motivações desta pesquisa para a construção de tal sistema:

- O fato de que jogar Damas representa um problema complexo e similar a vários problemas práticos da vida real [13];
- Opinião de Fogel [17] de que existe excessiva perícia humana na construção do campeão de Damas mundial *Chinook*;
- A oportunidade que os métodos TDs abrem no campo de habilidades da aprendizagem não supervisionada – mais detalhes veja seção 3.3.1.4;
- O fato que a primeira versão predecessora do *LS-VisionDraughts*, isto é, *LS-Draughts*, apesar de ser capaz de selecionar automaticamente um conjunto apropriado de *features* para representar os estados de tabuleiro na *MLP*, faz uso de um ineficiente algoritmo de busca para seleção da melhor ação – mais detalhes veja seção 3.3.1.2;
- O fato que a segunda versão predecessora do *LS-VisionDraughts*, isto é, *VisionDraughts*, apesar de contar com uma poderosa estratégia de busca, conta com um conjunto fixo de *features* escolhido manualmente e arbitrariamente para representar os estados de tabuleiro na *MLP* – mais detalhes veja seção 3.3.1.3.

Baseado nesses fatos, o sistema *LS-VisionDraughts* é então proposto: um agente não supervisionado que usa *AG* para automatizar a escolha das *features* do mapeamento *NET-FEATUREMAP*, aprendizagem TD para ajustar os pesos das redes *MLPs* e uma estratégia de busca baseada na poda *alfa-beta* combinada com tabela de transposição e aprofundamento iterativo para escolher o melhor movimento. A contribuição e objetivo desta proposta é analisar o ganho obtido pelo agente através da combinação da seleção automática de *features* do *LS-Draughts* com o eficiente módulo de busca do *VisionDraughts*,

sem contar com qualquer intervenção ou supervisão humana.

As próximas seções deste capítulo são organizadas da seguinte forma: seção 4.1 apresenta a arquitetura geral do *LS-VisionDraughts*; seção 4.2 detalha o módulo do *AG* responsável pela evolução das *features*; seções 4.3 e 4.4 apresentam, respectivamente, o módulo gerador das *MLPs* e o processo de treinamento das mesmas; seção 4.5 apresenta o módulo de busca do agente responsável por selecionar a melhor ação a ser executada a partir de um determinado estado de tabuleiro do jogo; por fim, seção 4.6 apresenta os resultados obtidos em torneio com o agente aqui proposto.

4.1 Arquitetura Geral do *LS-VisionDraughts*

A figura 12 apresenta a arquitetura geral do sistema híbrido *LS-VisionDraughts* através da *k-enésima* iteração de seu ciclo de evolução que pode ser resumido da seguinte forma: o *Módulo do AG* produz uma geração g_k cuja população é composta por T_p indivíduos, que representam subconjuntos de todas as *features* disponíveis do mapeamento *NET-FEATUREMAP*, e transfere-os para o *Módulo Gerador de MLPs*, como mostrado em #1 (passo 1 da figura 12). Esse módulo gera T_p novas redes *MLP*, onde cada *MLP* representa um indivíduo que será usado para mapear os estados de tabuleiro na entrada da *MLP* correspondente. As T_p redes *MLP* são transferidas ao *Módulo Treinamento* para serem treinadas individualmente, #2. As T_p *MLPs* treinadas são enviadas de volta ao *Módulo do AG*, #3. Esse módulo então usa essas *MLPs* para produzir a próxima geração g_{k+1}, fato que dispara a próxima iteração do ciclo mostrado na figura 12. Esse ciclo repete Q vezes, onde Q representa o número de gerações do *AG*. Os valores apropriados para T_p e Q são definidos considerando os seguintes cenários: *hardware* disponível para treinamento das *MLPs*, tempo razoável para treiná-las e a necessidade de obter, ao fim do *AG*, uma *MLP* capaz de tornar-se competitiva jogando contra outros agentes não supervisionados de Damas.

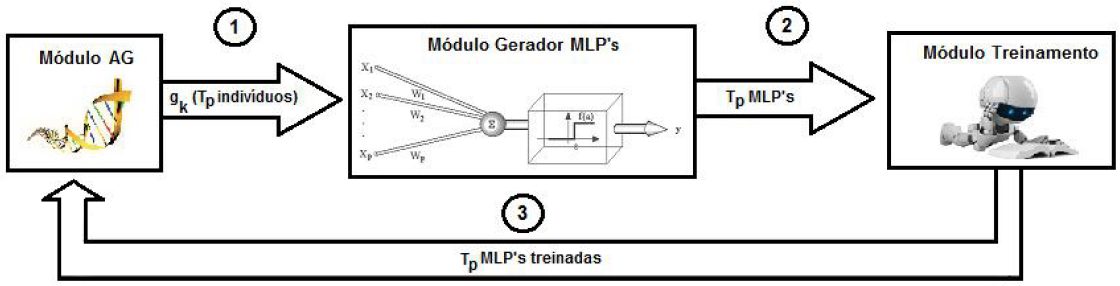

Figura 12 – Arquitetura geral do *LS-VisionDraughts*.

As próximas seções apresentam com mais detalhes cada um desses módulos e os valores adotados para T_p e Q.

4.2 O Módulo do AG

O principal papel do *Módulo do AG* é gerenciar a evolução da população de indivíduos do sistema *LS-VisionDraughts* ao longo de Q gerações, onde cada indivíduo corresponde a um subconjunto de todas as *features* disponíveis no mapeamento *NET-FEATUREMAP*. A tabela 6 mostra as principais características e parâmetros do *AG* utilizado pelo *LS-VisionDraughts*. Esses parâmetros são os mesmos utilizados pelo agente *LS-Draughts* em [12], exceto pelo fato que aqui foi considerado uma população de 40 indivíduos ao invés de 50. É importante destacar que o objetivo deste capítulo não é avaliar o sistema *LS-VisionDraughts* para diferentes parâmetros do *AG* (isto é, diferentes métodos de seleção, crossover, mutação, etc), mas apenas em obter o melhor indivíduo (melhor representação de tabuleiro do mapeamento *NET-FEATUREMAP*) que seja dotado de um eficiente mecanismo de busca na árvore do jogo, considerando o mesmo cenário evolutivo adotado em [12].

Tabela 6 – Parâmetros do AG utilizado pelo sistema *LS-VisionDraughts*.

Parâmetros	**V**alor
Geração	$Q = 30$
População	$T_p = 40$ indivíduos
Cromossomo	Binário com 15 genes
Seleção	Torneio Estocástico (tour = 3)
Crossover	Cruzamento simples (1 ponto de corte)
Taxa de Mutação	0.3 por indivíduo
Fitness	Torneio todos contra todos
Critério de Parada	Ao fim da 30ª geração

4.2.1 População e codificação dos indivíduos

Cada indivíduo da população é formado por um cromossomo fixo de 15 genes cuja representação binária indica se, em cada gene G_i, onde $i \in \{1, 2, 3, ..., 15\}$, há a presença (bit 1), ou não (bit 0), de uma determinada *feature* F_i referente ao mapeamento *NET-FEATUREMAP*. Considerando no exemplo da figura 13 que todos os genes de G8 a G13 estão preenchidos com valor binário 0, o indivíduo então representará o subconjunto $\{F1, F2, F5, F6, F7, F14\}$ de *features* do mapeamento *NET-FEATUREMAP*.

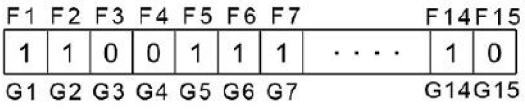

Figura 13 – Exemplo de codificação de um indivíduo na população.

A tabela 7 mostra as 15 *features* utilizadas pelo *Módulo do AG* para codificar os 15 genes de cada indivíduo. Essas 15 *features* do mapeamento *NET-FEATUREMAP* são

as mesmas utilizadas pelo *LS-Draughts* em [12]. É importante destacar que o sistema *LS-VisionDraughts* não trabalhou com todas as 26 *features* implementadas por Samuel em [13] devido à sua capacidade de processamento limitada – comprometida pelo fato de que o sistema não foi executado em cima de um ambiente de alta performance.

Conforme comentado na seção 2.6, cada número inteiro da coluna *BITS* da tabela 7, correspondente a uma *feature* F_i, indica a quantidade de neurônios que são reservados para representar F_i na camada de entrada da rede *MLP*.

Tabela 7 – *Features* do mapeamento *NET-FEATUREMAP* utilizadas pelo *Módulo do AG* do sistema *LS-VisionDraughts*.

Features	**B**its
F1: PieceAdvantage	4
F2: PieceDisadvantage	4
F3: PieceThreat	3
F4: PieceTake	3
F7: Backrowbridge	1
F8: Centrecontrol	3
F9: XCentrecontrol	3
F10: TotalMobility	4
F11: Exposure	3
F5: Advancement	3
F6: DoubleDiagonal	4
F12: KingCentreControl	3
F13: DiagonalMoment	3
F14: Threat	3
F15: Taken	3

A população do *AG* deste trabalho é composta por 40 indivíduos, isto é, $T_P = 40$. Portanto, a população será formada por 40 estruturas cromossômicas (ou indivíduos) que estarão associadas à 40 redes *MLPs* (cada rede para cada indivíduo). São esses 40 indivíduos que evoluirão dentro do *AG* ao longo de 30 gerações (Q=30). O processo de obtenção dos indivíduos em uma determinada geração do *AG* é definido da seguinte forma:

- Para a primeira geração, isto é, g_0, todos os 40 indivíduos são gerados aleatoriamente através da ativação binária aleatória (1 ou 0) de cada um de seus genes G_i, onde $i \in \{1, 2, 3, ..., 15\}$. A partir daí, uma rede *MLP* é acoplada a cada um dos 40 novos indivíduos gerados e o treinamento dessas redes é realizado. Em seguida, esses 40 indivíduos treinados são submetidos a um *torneio de avaliação* para o cálculo do *fitness* (detalhes desse torneio são apresentados na seção 4.2.3) e são passados como pais para a próxima geração, isto é, g_1;

- Em cada uma das 29 gerações restantes, isto é, g_k, onde $1 \leq k \leq 29$, os 40 novos indivíduos são gerados da seguinte forma: 20 pares de indivíduos são selecionados

dentre 40 pais recebidos da geração anterior g_{k-1} por meio do método de seleção torneio estocástico. Cada par gerará dois novos indivíduos através da aplicação dos operadores genéticos de *crossover* e mutação (detalhes sobre o método de seleção e aplicação dos operadores genéticos são apresentados na seção 4.2.2). Em seguida, 40 novas redes *MLP* são geradas e associadas a esses 40 novos indivíduos e então, treinadas. Depois o *Módulo do AG* inicia um torneio envolvendo os 40 novos indivíduos gerados e os 40 pais para o cálculo do *fitness*. Ao fim do torneio, os 40 melhores indivíduos, isto é, os indivíduos com os melhores valores de *fitness*, são passados como pais para a geração seguinte g_{k+1}. Esse processo repete-se até o fim da geração 29.

Finalmente, após a execução das 30 gerações, o melhor indivíduo, rede *MLP* com melhor *fitness*, obtido pelo processo evolutivo é, então, associado como o agente jogador do sistema *LS-VisionDraughts*.

4.2.2 Seleção dos indivíduos e aplicação dos operadores genéticos

O método de seleção utilizado pelo *LS-VisionDraughts* para selecionar os pais, a fim de aplicar os operadores genéticos que darão origem aos novos indivíduos, é o *torneio estocástico* com *tour* de 3 que opera da seguinte forma: dentre uma população de 40 pais, 3 indivíduos são selecionados pelo *método da roleta* e submetidos a um torneio, no qual o indivíduo ganhador é aquele que possuir o maior valor de *fitness*. Para cada 2 pais escolhidos pelo torneio estocástico, 2 novos filhos são gerados. O operador de *crossover* utilizado é o cruzamento simples de genes, isto é, *crossover* de um único ponto de corte, e a mutação é aplicada a todos os indivíduos da população. A taxa de probabilidade de mutação, P_{mut}, utilizada é de 0.3 por indivíduo, isto é, todos os indivíduos sofrem mutação em 5 genes escolhidos aleatoriamente.

A figura 13 mostra um exemplo de aplicação do operador de *crossover* com cruzamento simples na geração de dois novos indivíduos, filhos K e L, a partir de um par de indivíduos pais I e J. Na figura 14, exemplifica-se a aplicação do operador genético de mutação ao filho K, produzindo o indivíduo M.

4.2.3 Cálculo da função de adaptabilidade ou *fitness*

Para cada geração g_k, onde $1 \leq k \leq 29$, após o término de treinamento das 40 novas *MLPs*, que estão associadas aos 40 novos indivíduos gerados a partir da população de pais recebidos da geração g_{k-1}, um torneio é, então, realizado entre todos os 80 indivíduos, isto é, filhos e pais, com o objetivo de calcular suas *funções de adaptabilidade* ou *fitness*. O torneio consiste, basicamente, em totalizar a pontuação obtida por cada indivíduo I_i,

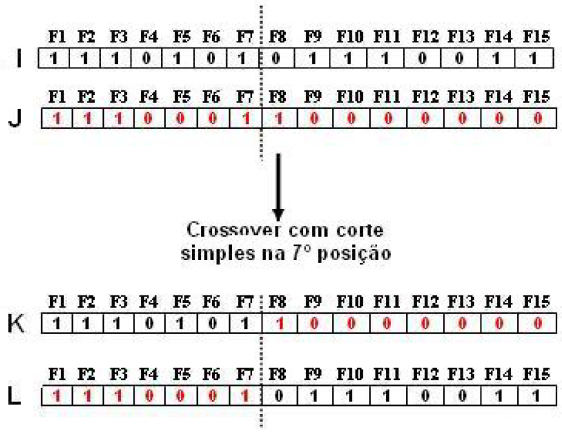

Figura 14 – Operação de *crossover*, com um único ponto de corte, aplicado a um par de indivíduos pais para gerar dois novos indivíduos.

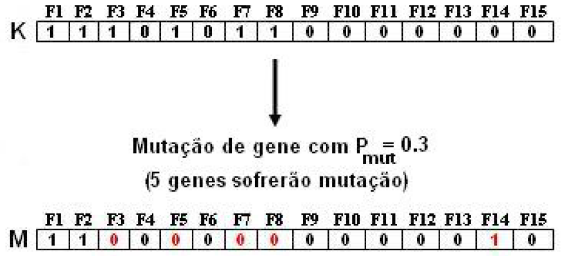

Figura 15 – Operação de mutação de gene com uma taxa de 0.3 sobre o indivíduo K.

onde $1 \leq i \leq 80$, em confrontos onde I_i joga 10 partidas contra todos os demais indivíduos da geração g_k. Assim, o *fitness* calculado para I_i será obtido em função dos resultados de 790 partidas disputadas. A pontuação por jogo é definida da seguinte forma: 2 pontos por vitória, 1 ponto por empate e 0 por derrota. O objetivo dos 10 jogos disputados em cada confronto é avaliar o desempenho do indivíduo I_i ao jogar alternadamente, como jogador preto e branco, em 5 jogos com aberturas iniciais diferentes do tabuleiro de Damas. Por exemplo, suponha que o desempenho de um indivíduo I_2 em um torneio de 790 jogos disputados na geração g_{10} seja o seguinte: 470 vitórias, 255 empates e 65 derrotas. Nesse caso, o *fitness* de I_2 será 1.195 pontos.

Vale destacar que para o caso especial da geração inicial g_0, o torneio é realizado sempre entre 40 indivíduos (população inicial), ao invés de 80 indivíduos como acontece nas demais gerações.

4.3 O Módulo Gerador de MLPs

Após a geração de um novo indivíduo I_i, por qualquer uma das duas formas descritas na seção 4.2.1, onde $i \in \{1, 2, 3, ..., 40\}$, um filtro é aplicado à sua sequência de genes selecionando apenas as *features* que estão presentes ou ativas, isto é, com bit 1. A partir daí, uma rede neural *MLP* com N_A neurônios na camada de entrada, 20 neurônios na camada oculta e um único neurônio na camada de saída é gerada e acoplada ao novo indivíduo I_i. A variável N_A representa a quantidade de bits associados com os genes ativos ou *features* ativas. Por exemplo, considerando na figura 16 que todos os genes de G4 a G13 estão preenchidos com valor binário 0, a rede acoplada ao indivíduo M, que tem somente 3 genes ativos, utilizará as *features* ativas F_1, F_2 e F_{14} para representar o tabuleiro de Damas na entrada da rede *MLP*. Considerando os valores apresentados na tabela 7 da seção 4.2.1, N_A é igual a 11, isto é, a camada de entrada da *MLP* associada ao indivíduo M será formada por 11 neurônios.

Os pesos iniciais da rede *MLP* vinculada ao indivíduo I_i são gerados aleatoriamente entre $-0,2$ e $+0,2$ e o termo *bias* é fixado em -1. Esse processo repete-se para todos indivíduos I_i, onde $1 \leq i \leq 40$.

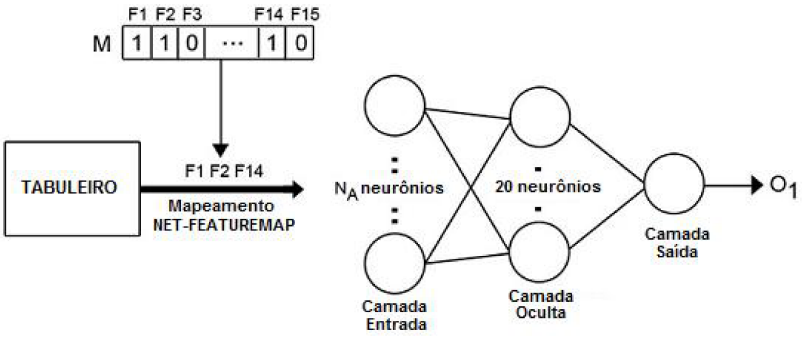

Figura 16 – Geração de uma *MLP* a partir de um cromossomo ou indivíduo.

4.4 O Módulo de Treinamento

No módulo de treinamento, dois tipos de jogos são considerados sobre as *MLPs* do sistema *LS-VisionDraughts*: *jogos treino* e *jogos não treino*. Como o objetivo do primeiro grupo é treinar as *MLPs*, durante esses jogos os pesos das redes são continuamente atualizados. Por outro lado, os *jogos não treino* correspondem às competições normais entre o agente e seus oponentes, isto é, sem atualização dos pesos.

A figura 17 mostra o processo de aprendizagem das redes *MLPs* do sistema *LS-VisionDraughts* durante os *jogos treino*. Resumidamente, sempre que o agente precisa escolher um novo movimento, aqui denominado por a_{t+1}, pois $t+1$ representa um tempo futuro ao estado ou tempo corrente t, o ciclo do processo recursivo mostrado na figura

17 é acionado. O estado de tabuleiro corrente, aqui denominado por S_t, cuja predição P_t foi calculada no ciclo anterior relacionado à escolha do último movimento, é apresentado ao *Módulo de Busca* #1 (passo 1 na figura). Esse módulo monta uma árvore de busca do jogo cuja raiz é o estado S_t. Cada estado de tabuleiro associado aos nós folha da árvore de busca é convertido na representação *NET-FEATUREMAP*, #2, e apresentado para a camada de entrada da *MLP*, #3. A *MLP* então avalia cada um desses nós folha e retorna um valor (*predição*) que indica o quanto o estado nó folha é favorável para o agente. Esse valor é retornado para o *Módulo de Busca*, #4, de forma que o algoritmo de busca *alfa-beta* possa indicar o melhor movimento a_{t+1} a ser executado em S_t, #5. O agente então executa esse movimento, #6. O novo estado de tabuleiro S_{t+1} é convertido na representação *NET-FEATUREMAP*, #7, e apresentado para a *MLP* avaliá-lo gerando uma predição P_{t+1}, #8. As predições P_{t+1} referente ao estado S_{t+1} (novo estado) e P_t referente ao estado S_t (antigo estado corrente cuja predição foi calculada no ciclo anterior) são usadas pelo *Módulo de Aprendizagem TD* como parâmetros de entrada do método TD(λ) para ajustar os pesos da rede *MLP*, #9 e #10. Em seguida, o estado S_{t+1} é avaliado novamente pela rede *MLP*, agora com os pesos já reajustados, #11, gerando um segundo valor de predição P'_{t+1}, #12. Esse segundo valor instanciará a variável P_t do ciclo de treinamento da figura 17, #13, isto é, o novo estado S_{t+1}, produzido pela ação a_{t+1} sugerida pelo *Módulo de Busca*, passa a ser o próximo estado corrente S_t do ciclo de treinamento da figura, #14, com predição P_t que recebeu a nova predição P'_{t+1} obtida nas etapas #11 e #12.

Durante os *jogos não treino*, o processo é análogo ao ciclo de treinamento apresentado na figura 17, exceto nos seguintes aspectos: o *Módulo de Aprendizagem TD* e os passos #9, #10, #11 e #12 devem ser desconsiderados. Além disso, na etapa #13 a variável P_t é instanciada com a própria predição P_{t+1} referente ao novo estado S_{t+1}, pois nesse caso não há geração de um novo valor P'_{t+1} que só ocorre quando a rede está treinando.

As próximas seções descrevem com mais detalhes o processo de atualização dos pesos das redes *MLPs* e a estratégia de treinamento por *self-play* com clonagem adotada pelo sistema *LS-VisionDraughts*. Em seguida o *Módulo de Busca* é apresentado.

4.4.1 Atualização dos Pesos das MLPs

Esta seção explica como o processo de reajuste ou atualização dos pesos de cada *MLP* é realizado pelo *LS-VisionDraughts* para aprender a jogar Damas. Considerando um conjunto de movimentos que o agente executa durante os *jogos treino*, isto é, a_0,..., a_{i-1}, a_i, a_{i+1}, a_f, onde a_f refere-se a ação que conduz o tabuleiro para o estado final do jogo, um reforço final é fornecido pelo ambiente à *MLP*, de acordo com o resultado obtido pelo agente em função da sequência de movimentos executados. O reforço é positivo, valor +1, caso o resultado do jogo tenha sido vitória, negativo, valor -1, caso o resultado tenha sido derrota ou zero (0) caso tenha ocorrido um empate.

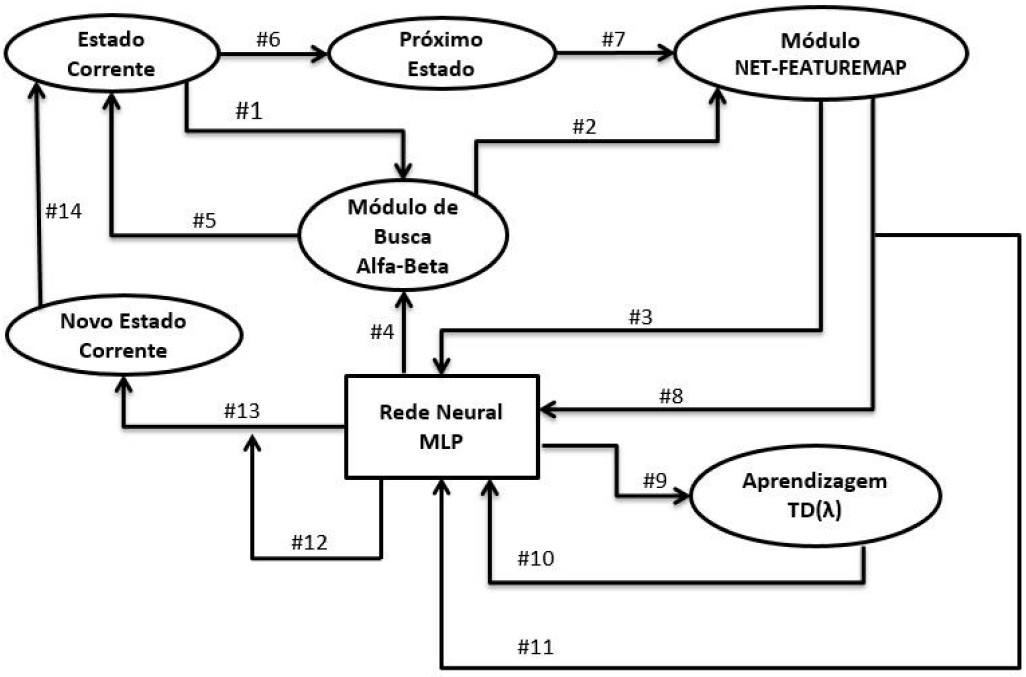

Figura 17 – Processo de Aprendizagem do *LS-VisionDraughts*.

Conforme apresentado na figura 17 da seção 4.4, antes do agente executar qualquer movimento sobre um tabuleiro corrente S_t, uma árvore de busca é montada com raiz em S_t com o objetivo de obter, pelo processo de busca e utilizando os pesos atuais da *MLP*, qual a melhor ação a_{t+1} (ação futura) a ser executada em S_t. O estado resultante, S_{t+1}, é então mapeado na entrada da rede *MLP* e tem sua predição P_{t+1} calculada. Calcular a predição P_{t+1} para o estado S_{t+1} do jogo de Damas implica em apresentar tal estado, convertido no mapeamento *NET-FEATUREMAP*, às unidades da camada de entrada da *MLP* e, a partir dessa camada, calcular os campos locais induzidos e os sinais funcionais da rede prosseguindo para frente, camada por camada, até obter o valor de saída O_{t+1} que representa a saída do neurônio da última camada *MLP*. Resumidamente, a predição P_{t+1} é uma função dependente do vetor de entrada $\overrightarrow{X}(t+1)$, vetor obtido pela conversão do estado S_{t+1} em um vetor de bits do mapeamento *NET-FEATUREMAP*, e do vetor de pesos $\overrightarrow{W}(t+1)$ da *MLP* no instante temporal $t+1$, isto é, $P_{t+1}(\overrightarrow{X}(t+1), \overrightarrow{W}(t+1))$. Para mais detalhes sobre o cálculo de predição de uma *MLP*, veja [11].

Assim, para cada duas predições temporais sucessivas P_t, predição associada ao "antigo" estado corrente S_t, e P_{t+1}, predição associada ao "novo" estado corrente S_{t+1}, o mecanismo TD então ajusta os pesos da *MLP* através da equação do método TD(λ) [79]:

$$\begin{aligned} w_{ij}^{(l)} &= w_{ij}^{(l)}(t) + \Delta w_{ij}^{(l)}(t+1) \\ &= w_{ij}^{(l)}(t) + \alpha^{(l)} \cdot (P_{t+1} - P_t) \cdot \sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k \\ &= w_{ij}^{(l)}(t) + \alpha^{(l)} \cdot (P_{t+1} - P_t) \cdot elig_{ij}^{(l)}(t), \text{ onde:} \end{aligned} \tag{4}$$

- $\alpha^{(l)}$ é o parâmetro da taxa de aprendizagem na camada l. Foi utilizado 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 $t+1$ é representada por $\Delta w_{ij}^{(l)}(t+1)$;

- O termo $elig_{ij}^{(l)}(t)$ é único para cada peso sináptico $w_{ij}^{(l)}(t)$ da rede neural e representa o rastro de eligibilidade das predições calculadas pela rede para os estados resultantes dos movimentos executados pelo agente desde o instante temporal 1 do jogo até o instante temporal t. O rastro de elegibilidade é um dos mecanismos básicos utilizados na AR para lidar com recompensas "atrasadas" (do inglês *delayed rewards*). Para mais detalhes sobre o rastro de eligibilidade, veja [100];

- $\nabla_w P_k$ 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 $0 \leq \lambda \leq 1$, tem o papel de dar uma "pesagem exponencial" para a taxa de variação das predições calculadas a k passos anteriores de t. Quanto maior for λ, maior o impacto dos reajustes anteriores ao instante temporal t sobre o reajuste dos pesos $w_{ij}^{(l)}(t)$.

O processo de reajuste dos pesos por Diferenças Temporais TD(λ) são descritos nas seguintes etapas:

1. O vetor $\overrightarrow{W}(k)$ de pesos é inicializado 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 movimentos executados pelo agente durante o jogo, define-se o sinal de erro pela equação:

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

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

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

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

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

em que λ tem o papel de fornecer uma "pesagem exponencial" para a taxa de variação das predições calculadas a k passos anteriores de t; $a_i^{(l)}$ representa o sinal de saída do neurônio i na camada l; $g'(x) = (1-x^2)$ representa a derivada da função de ativação utilizada aqui, tangente hiperbólica;

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

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

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

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

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

5. Calculadas 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:

$$\Delta w_{ij}^{(l)}(t+1) = \alpha^{(l)} \cdot e(t+1) \cdot elig_{ij}^{(l)}(t), \tag{5}$$

onde o parâmetro de aprendizagem $\alpha^{(l)}$ é definido 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 *MLPs*, que é o fato de a convergência estar assegurada para um mínimo local do erro e não necessariamente para o mínimo global do erro. Quando a superfície de erro é boa isto não representa um problema, mas quando a superfície apresenta muitos mínimos locais, a convergência não é assegurada para o melhor valor. Nesses casos, geralmente utiliza-se o termo momento μ para tentar solucionar esse tipo de problema. A adição do termo momento no método TD(λ) determina o efeito das mudanças anteriores dos pesos na direção atual do movimento no espaço de pesos. Em outras palavras, o termo momento evita que o equilíbrio da função de avaliação se estabeleça em regiões cujo erro mínimo seja sub-ótimo [40]. Para resolver esse problema foi empregado uma checagem de direção na equação 4, ou seja, o termo momento μ é aplicado somente quando a correção do peso atual $\Delta w_{ij}^{(l)}(t+1)$ e a correção anterior $\Delta w_{ij}^{(l)}(t)$ estiverem na mesma direção. Portanto, a equação final TD(λ) utilizada para calcular o reajuste dos pesos da rede neural na camada l, para $0 \leq l \leq 1$, é definida por:

$$w_{ij}^{(l)}(t+1) = w_{ij}^{(l)}(t) + \Delta w_{ij}^{(l)}(t+1); \tag{6}$$

onde $\Delta w_{ij}^{(l)}(t+1)$ é obtido nas seguintes etapas:

a) calcule $\Delta w_{ij}^{(l)}(t+1)$ pela equação 4;

b) se ($\Delta w_{ij}^{(l)}(t+1) > 0$ e $\Delta w_{ij}^{(l)}(t) > 0$) ou ($\Delta w_{ij}^{(l)}(t+1) < 0$ e $\Delta w_{ij}^{(l)}(t) < 0$), então faça:

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

4.4.2 Estratégia de treinamento por *self-play* com clonagem

A estratégia de treinamento adotada pelo *LS-VisionDraughts* nos *jogos treino* de suas redes *MLPs* é a mesma proposta por Lynch em [40], isto é, estratégia de treinamento por *self-play* com *clonagem*. Tal estratégia consiste em treinar um agente jogador (*MLP*) por vários jogos contra uma cópia de si próprio (daí o nome *self-play*). À medida que o jogador aumenta seu desempenho ao ponto de conseguir vencer sua cópia, uma nova *clonagem* é realizada e o jogador passa a treinar contra esse novo *clone*. O processo repete-se por um determinado número de *jogos treino*.

No treinamento por *self-play* com clonagem, cada *MLP* é submetida a 4 sessões de 400 jogos de treinamento (*jogos treino*), onde a rede joga metade desse jogos (200) com as peças pretas do tabuleiro de Damas e a outra metade com as peças brancas. Essa estratégia do agente trocar as cores das peças do tabuleiro durante o treinamento é interessante por 2 motivos: treinar o agente para jogar em ambas as situações e porque algumas *features* do mapeamento *NET-FEATUREMAP* estabelecem restrições relativas às cores das peças do tabuleiro, fato que tende a beneficiar o agente que joga com tais cores. Por exemplo, a *feature Backrowbridge* contabiliza a quantidade de peças pretas nas posições 1 e 3 do tabuleiro, como também a não existência de damas brancas no tabuleiro todo.

Resumidamente, a estratégia de treinamento por *self-play* com clonagem de uma determinada *MLP*, aqui denominada por RNA_i, pois refere-se a um determinado indivíduo I_i da população do AG, onde $i \in \{1, 2, 3, ..., 40\}$, pode ser divida nas seguintes etapas:

1. Primeiro, os pesos da RNA_i são gerados aleatoriamente;

2. Antes de iniciar a primeira sessão de treinamento, a rede RNA_i é clonada, isto é, uma cópia exata da rede, incluindo arquitetura e pesos sinápticos, é realizada, gerando o primeiro clone RNA_i-*clone*;

3. Inicia-se então a primeira sessão de 400 jogos de treinamento entre as redes RNA_i e RNA_i-*clone*. Durante essa sessão de treinamento somente os pesos da rede RNA_i são ajustados conforme processo apresentado nas seções 4.4 e 4.4.1;

4. Ao final da primeira sessão de treinamento, dois *jogos teste* são realizados para verificar qual das redes é a melhor. Caso a rede RNA_i (rede com reajuste de pesos) supere a rede clonada RNA_i-*clone* (rede sem reajuste de pesos), então é realizada uma nova clonagem (cópia) da rede RNA_i para a rede RNA_i-*clone*. Caso contrário,

os pesos da rede clonada RNA_i-*clone* permanecem os mesmos para a próxima sessão de treinamento;

5. Volte para etapa 3 e execute uma nova sessão de 400 jogos de treinamento entre a rede RNA_i e o seu último clone RNA_i-*clone*. Repita o processo até que o número máximo de 4 sessões de treinamento seja alcançado;

6. Ao final de todas as 4 sessões de treinamento, realiza-se um torneio de dois *jogos teste* entre a ultima versão da rede RNA_i, obtida na última sessão de treinamento, e todos os seus clones realizados. Finalmente, o vencedor desse torneio é considerada a melhor *MLP* para representar o indivíduo I_i. Tal estratégia é utilizada por este trabalho para evitar o problema de *sobre-ajuste* (do inglês *overfitting*) de pesos da rede *MLP* que está em processo de treinamento.

4.5 O processo de Busca

Conforme comentado na seção 2.4, em agentes jogadores que adotam o algoritmo *Minimax* para escolher o melhor movimento, a capacidade de realizar pesquisas mais profundas na árvore de busca é seriamente comprometida, pois tal algoritmo não é capaz de realizar podas na árvore do jogo e consequentemente, examina mais estados de tabuleiro do que o necessário. Testes envolvendo os agentes *NeuroDraughts* e *LS-Draughts*, agentes que utilizam o algoritmo *Minimax*, mostram que é inconveniente que a profundidade máxima de suas buscas ultrapassem uma profundidade 4 [27]. Por essa razão, no *LS-VisionDraughts*, o *Minimax* é substituído por uma versão modificada do clássico algoritmo de busca *Alfa-Beta* combinado com TT e aprofundamento iterativo, além de poder contar, opcionalmente, com bases de final de jogo, conforme é explicado nas próximas seções.

Os resultados experimentais apresentados na seção 4.6 confirmam a melhora obtida pelo agente com essa nova estratégia de busca, combinada com *AG*, em relação às versões predecessoras – no quesito qualidade e tempo de execução.

4.5.1 Adaptando a poda *Fail-Soft Alfa-Beta* para o Agente

O algoritmo *Alfa-Beta* pode ser resumido como um procedimento recursivo que escolhe o melhor movimento a ser executado realizando uma busca em profundidade, da esquerda para a direita, na árvore do jogo. Existem duas versões do algoritmo *Alfa-Beta*, *fail-soft* e *hard-soft*, as quais diferem-se pelo valor de predição retornado para um estado quando ocorre uma poda. A versão *fail-soft* sempre retorna o valor real da predição para o estado avaliado, independente se houve ou não poda. Já a versão *hard-soft* retorna como predição para o estado, o valor correspondente ao limite da janela (*alfa* ou *beta*) que efetuou a poda, desconsiderando sua predição real. Essa particularidade da versão *hard-soft* restringe sua combinação com a TT, uma vez que o valor de predição de um

estado pode não corresponder a predição real para o mesmo, no caso de ocorrer poda. Tal fato, faz com que os movimentos executados pelo algoritmo *hard-soft* não sejam compatíveis quando comparado com sua versão sem a TT, e assim, inviabilizam sua combinação com a TT. Dessa maneira, como a TT compõe o módulo de busca do agente, a versão utilizada nesse trabalho é a *fail-soft* (detalhes sobre a inserção da TT no módulo de busca são apresentados na seção 4.5.2).

No *LS-VisionDraughts*, a evolução do jogo de Damas é representado por uma árvore de possíveis estados de tabuleiros, sendo que cada nó representa um estado do jogo e cada ramo representa um possível movimento [101]. Cada vez que o agente precisa executar um movimento, o algoritmo *Alfa-Beta* é usado para construir a árvore de busca e escolher o melhor movimento. A figura 18 mostra um exemplo de expansão da árvore do jogo realizada pela versão *fail-soft* do algoritmo *Alfa-Beta*. A raiz da árvore de busca corresponde ao estado corrente do jogo S_1 – o estado S_1 corresponde ao estado S_t do processo de aprendizagem da *MLP* explicado na seção 4.4. O próximo nível da árvore, isto é, profundidade 1, corresponde aos possíveis estados que podem ser obtidos a partir de movimentos válidos executados em S_1. Os nós dos níveis mais profundos da árvore são obtidos de forma similar até que o nível de profundidade máxima d seja obtido. Conforme será explicado na seção 4.5.6, a profundidade máxima d é controlada pelo tempo – parâmetro *max-time* – ou pela própria profundidade – parâmetro *max-depth*. A *MLP* então calcula a predição P para cada estado pertencente à profundidade d da árvore de busca e, essas predições são retornadas para o algoritmo de busca, que indica ao agente o melhor movimento a ser executado em S_1.

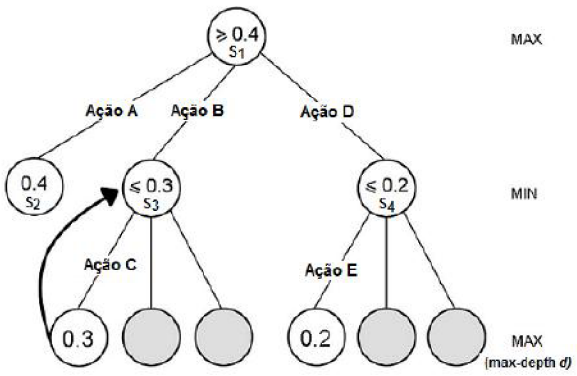

Figura 18 – Árvore do jogo expandida pelo algoritmo *Fail-Soft Alfa-Beta*.

Os tópicos a seguir explicam com mais detalhes cada etapa do pseudo-código do algoritmo 2 referente à versão *Fail-Soft Alfa-Beta*.

- **Linha 1:** para a escolha do melhor movimento, os seguintes parâmetros de entrada são fornecidos ao *Alfa-Beta*: o estado corrente do jogo, S_i, a profundidade d de busca, o valor *alfa* que representa o limite inferior do intervalo de busca e o valor *beta* que representa o limite superior do intervalo de busca; *alfa* e *beta* são inicializados

Algoritmo 2 :Pseudo-código da versão *fail-soft* do algoritmo *Alfa-Beta*

```
 1: function alfaBeta(node:S_i, int:d, int:alfa, int:beta, move:bestmove)
 2: method:
 3: if leaf(S_i) or d=0 then
 4:    return evaluate(S_i)
 5: end if
 6: if S_i is a max node then
 7:    besteval := alfa
 8:    for each child of S_i: do
 9:       v := alfaBeta(child,d-1,besteval,beta,bestmove)
10:       if v > besteval then
11:          besteval:= v
12:          thebest = bestmove
13:       end if
14:    end for
15:    if besteval >= beta then
16:       return besteval
17:    end if
18:    bestmove = thebest
19:    return besteval
20: end if
21: if S_i is a min node then
22:    besteval := beta
23:    for each child of S_i: do
24:       v := alfaBeta(child,d-1,alfa,besteval,bestmove)
25:       if v < besteval then
26:          besteval:= v
27:          thebest = bestmove
28:       end if
29:    end for
30:    if besteval <= alfa then
31:       return besteval
32:    end if
33:    bestmove = thebest
34:    return besteval
35: end if
```

com os valores $-\infty$ e $+\infty$, respectivamente, e seus valores vão sendo atualizados a medida que os nós folhas da árvore de busca vão sendo avaliados. O parâmetro de saída *bestmove* corresponde ao melhor movimento a_i a ser executado sobre o estado corrente S_i. Para o exemplo da figura 18, *bestmove* corresponde ao movimento "Ação A" a ser executado sobre o estado corrente S_1.

- **Linhas 3 a 5:** por tratar-se de um procedimento recursivo, exige-se uma condição de parada: verificar se o estado do tabuleiro S_i é uma folha da árvore, ou seja, se não possui filhos. Nesse caso, a função *evaluate*(S_i) retorna a predição dada pela *MLP* para o estado S_i;

- **Linha 6:** testa se o estado S_i corresponde a um nó maximizador, então executa as linhas 7 a 20, a serem detalhadas na sequência;

- **Linha 7:** *besteval* representa a melhor avaliação encontrada para o nó S_i até o

presente momento. Como S_i representa um nó maximizador, inicialmente o valor de *besteval* é configurado como o maior valor negativo possível. Note que *besteval* será incrementado até o máximo valor das predições associadas aos filhos de S_i (linhas de 8 a 14);

- **Linhas 8 a 14:** para cada um dos filhos *child*, do estado S_i, o algoritmo *Alfa-Beta* é chamado, recursivamente, com profundidade *d - 1*. O intervalo de busca utilizado para a chamada recursiva será [*besteval; beta*] (linha 9). No nível de maximização, sempre que ocorrer atualização de um dos limites do intervalo de busca, a atualização acontecerá no limite inferior (no caso, em *besteval*). Isso acontece sempre que a predição v calculada para *child* superar o valor de *besteval* (linha 10). Caso a predição de v seja maior que *besteval*, melhor avaliação encontrada até o momento para S_i, o algoritmo atualiza *besteval* com o valor de v e o valor de *thebest* (linhas 11 e 12). O valor de *thebest* corresponde ao melhor movimento (*bestmove*) encontrado até o momento para S_i;

- **Linhas 15 a 17:** se acontecer da predição armazenada em *besteval* ultrapassar o limite superior do intervalo de busca ($besteval \geq beta$), o algoritmo retornará, imediatamente, o valor de *besteval* como predição associada ao estado S_i. Tal fato expressa a ideia de que a predição associada ao estado S_i é, no mínimo, *besteval*;

- **Linhas 18 a 20:** após a avaliação do todos os filhos *child* do estado S_i, *thebest* conterá o melhor movimento a ser executado a partir do estado S_i; *besteval* conterá a maior predição de todos os filhos do estado S_i, pois S_i é maximizador, e será retornado como valor da predição associada ao estado S_i. Novamente, no exemplo da figura 18, o estado raiz S_1 (nó maximizador) terão como valores $besteval = 0.4$ e $thebest =$ "*AÇÃO A*";

- **Linhas 21 a 35:** testa se o estado S_i corresponde a um nó minimizador (linha 21), então executa o mesmo processo análogo descrito entre as linhas 6 e 20, porém do ponto de vista de um nó minimizador. Nesse caso, a predição v associada ao nó S_i será igual à *menor* predição dos seus filhos e o intervalo de busca utilizado será [*alfa; besteval*] (linha 24). No nível de minimização, sempre que ocorrer atualização de um dos limites do intervalo de busca, ela será um decremento no limite superior. Isso acontece sempre que a predição v calculada para *child* for inferior ao valor de *besteval* (linha 25). Caso a predição de v seja menor que *besteval*, melhor avaliação encontrada até o momento para S_i, o algoritmo atualiza *besteval* com o valor de v e o valor de *thebest* (linhas 26 e 27). Se acontecer da predição armazenada em *besteval* ser menor que o limite inferior do intervalo de busca, isto é, $besteval \leq alfa$ (linha 30), o algoritmo retornará, imediatamente, o valor de *besteval* como predição associada ao estado. Tal fato expressa a ideia de que a predição associada ao estado

é, no máximo, *besteval*. Após a avaliação do todos os filhos *child* do estado S_i, *thebest* conterá o melhor movimento a ser executado a partir do estado S_i (linha 33); *besteval* conterá a menor predição de todos os filhos do estado S_i, pois S_i é minimizador, e será retornado como valor da predição associada ao estado S_i (linha 34). Novamente, no exemplo da figura 18, o estado S_3 (nó minimizador) terão como valores *besteval* = 0.3 e *thebest* = "*AÇÃO C*";

4.5.2 Aplicação da TT no Módulo de Busca

O algoritmo *Fail-Soft Alfa-Beta* apresentado na seção 4.5.1 não mantém um histórico dos estados da árvore do jogo procurados anteriormente. Assim, se um estado S_i qualquer do tabuleiro for apresentado, por exemplo, 2 vezes para o algoritmo *Alfa-Beta*, a mesma rotina será executada 2 vezes a fim de encontrar a predição associada a S_i. Esse tipo de comportamento é ainda mais comum em estratégias de busca com *aprofundamento iterativo* (tal estratégia será apresentada com mais detalhes na seção 4.5.6). Para evitar esse tipo de redundância, é recomendado o uso de *tabelas de transposição* ou *repositórios de predições passadas* associadas aos estados de tabuleiro do jogo que já foram submetidos ao procedimento de busca. Além disso, particularmente em alguns jogos, como por exemplo Damas e Xadrez, dentro de uma mesma partida, pode-se chegar a um mesmo estado de tabuleiro várias vezes e, quando isso ocorre, diz-se que houve uma transposição, daí a origem do nome *tabela de transposição* [76].

No *LS-VisionDraughts* a TT utilizada para armazenar informações relevantes sobre os estados de tabuleiro, que já foram explorados pelo algoritmo de busca, é a tabela *hash*. Para isso, os tabuleiros do jogo são representados na forma de *chaves hash* seguindo a técnica de Zobrist [101] e [102]. Tal técnica usa uma sequência de bits aleatórios distintos de comprimento fixo, chamado *chave Zobrist*, para representar as 32 posições do tabuleiro de Damas. Como cada posição do tabuleiro pode representar 5 estados diferentes: peça simples preta ou branca, dama preta ou branca e vazio, então, a *chave Zobrist* para Damas requer 32 x 4 = 128 entradas – observe que as posições vazias não precisam ser representadas. A figura 19 mostra as 128 *chaves Zobrist* composta de uma sequência fixa de 64 bits para cada uma das 32 posições do tabuleiro de Damas.

Com o objetivo de garantir a qualidade dos números aleatórios gerados para as *chaves Zobrist* apresentadas na figura 19 (coluna *"RANDOM INT64"* da figura), tais inteiros foram gerados a partir do gerador de bits aleatórios [103], utilizando a técnica descrita em [104], o que garante a aleatoriedade da sequência gerada baseando-se na aleatoriedade intrínseca de processos físicos em que fótons são detectados ao acaso [27], [76].

Utilizando a *chave Zobrist*, o processo de criação de uma *chave hash*, que será atribuído a um determinado tabuleiro do jogo de Damas, é dado da seguinte forma [76]:

1. Considere um estado S_i do tabuleiro do jogo de damas como sendo o mostrado na

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

Figura 19 – Vetor de 128 elementos inteiros aleatórios com as *chaves Zobrist* utilizadas pelo *LS-VisionDraughts*.

figura 20;

2. Para conseguir o número aleatório associado à peça preta simples localizada na posição 2 do tabuleiro S_i, basta fazer uma consulta ao vetor da figura 19 e encontrar a *chave Zobrist* = 17903615704209920410 (valor da coluna *"RANDOM INT64"* da figura referente à primeira linha do *SQUARE* 2);

3. Para conseguir o número aleatório associado à dama preta, localizado na casa 31 do tabuleiro S_i, basta fazer uma consulta ao vetor da figura 19 e encontrar a *chave Zobrist* = 14507257672045050736 (valor da coluna *"RANDOM INT64"* da figura referente à terceira linha do *SQUARE* 31).

4. Para conseguir o número aleatório associado à dama branca, localizado na casa 3 do tabuleiro S_i, basta fazer uma consulta ao vetor da figura 19 e encontrar a *chave Zobrist* = 3651659200175719294 (valor da coluna *"RANDOM INT64"* da figura referente à quarta linha do *SQUARE* 3).

5. Assim, para conseguir a *chave hash* H_i associada ao estado do tabuleiro S_i, aplique o operador XOR nas três *chaves Zobrist* obtidas nas etapas 2, 3 e 4, isto é, $H_i = 17903615704209920410 \oplus 14507257672045050736 \oplus 3651659200175719294$, ou seja, $H_i = 17159320952789611153$.

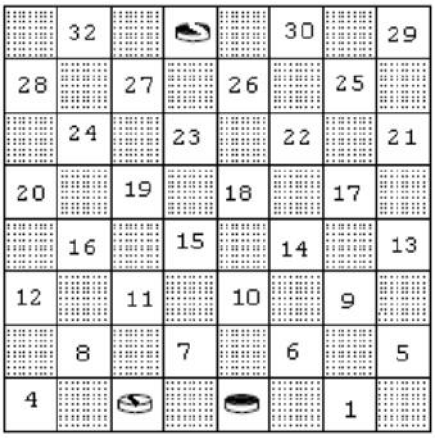

Figura 20 – Um estado do tabuleiro do jogo de damas.

A técnica de Zobrist é, provavelmente, o método mais rápido disponível para calcular uma *chave hash* associada a um estado de tabuleiro do jogo de Damas, já que a velocidade com que uma operação XOR é executada por CPU é extremamente rápida, além de possibilitar uma atualização incremental na *chave hash* com movimentações de peças sobre um tabuleiro [27], [76].

4.5.3 Tratamento do Problema de Colisão na TT

A TT do *LS-VisionDraughts* trata dois problemas de colisão identificados por Zobrist em [102]: *erro tipo 1* e *erro tipo 2*. O *erro tipo 1*, também conhecido por *clash*, ocorre quando dois estados distintos de tabuleiro de Damas são mapeados com a mesma *chave hash*. Se tal erro não for tratado adequadamente, pode acontecer de predições incorretas serem retornadas pela rotina de busca *Alfa-Beta* ao consultar a TT. Para reduzir a probabilidade de ocorrência de *erro tipo 1*, *LS-VisionDraughts* usa duas *chaves hash*: *hashvalue* de 64 bits mostrado na seção 4.5.2 e *checksum* de 32 bits, que segue a mesma lógica do *hashvalue* de 64 bits, que praticamente eliminam as ocorrências de *clashes* (para mais detalhes sobre tais chaves, veja [76]).

O segundo tipo de erro tratado no *LS-VisionDraughts* é a colisão de *erro tipo 2*. Ele acontece quando dois estados de tabuleiro distintos, apesar de serem mapeados com *chaves hash* diferentes, são direcionados para o mesmo endereço na TT. Isso ocorre devido a limitação de memória disponível para a TT e o enorme espaço de estados inerente ao jogo de Damas. É por isso que *LS-VisionDraughts* usa a operação modular (*mod*), entre a *chave hash* e a quantidade de entradas reservadas em memória para a TT, para definir

o endereço de memória que o tabuleiro associado a *chave hash* será armazenado. Mas mesmo assim, tal operação não impede que colisões de *erro tipo 2* aconteçam. Para controlar esse tipo de colisão, *LS-VisionDraughts* baseia-se em dois *esquemas de substituição* (do inglês *replacement schemes*), chamados *Deep* e *New*, propostos por Breuker em [105]. De acordo com Breuker, se uma TT tiver dois níveis, isto é, capacidade de armazenar informações referentes a dois estados de tabuleiro no mesmo endereço, ela terá melhor performance do que uma outra TT com o dobro de capacidade de armazenamento, mas com apenas um nível de armazenamento.

Mais especificamente, de acordo com a estratégia *Deep* e *New* proposta por Breuker, na primeira vez que um determinado endereço é selecionado para armazenar alguma informação de um determinado estado S_i, ele será gravado no primeiro nível. O segundo nível só será usado se ocorrer alguma colisão de *erro tipo 2*, que é quando é utilizado o esquema de substituição *New*, ou em caso do primeiro nível já ter informação armazenada para o estado S_i. Nesse caso, o primeiro nível sempre mantem a informação mais precisa do estado S_i utilizando o esquema de substituição *Deep*. Nesse esquema, a predição calculada para S_i referente a sub-árvore mais profunda é preservada. Portanto, sempre que um determinado endereço de entrada na TT armazena informações referentes ao mesmo estado de tabuleiro S_i em ambos os níveis, isto significa que a informação armazenada no primeiro nível foi obtida a partir de uma posição mais profunda na árvore do jogo, ou seja, informação mais precisa. Por outro lado, se o mesmo endereço de entrada na TT armazena informações referentes a dois estados diferentes, isto significa que o segundo nível foi utilizado para resolver o problema de colisão de *erro tipo 2* (para mais detalhes sobre tais *esquemas de substituição*, veja [76]).

É importante destacar que com a aplicação dos dois esquemas de substituição de Breuker, *Deep* e *New*, foi possível observar, experimentalmente, que os problemas de colisão de *erro tipo 2* foram totalmente controlados, conforme procedimento explicado acima, no *LS-VisionDraughts* [27].

4.5.4 Armazenando Estados de Tabuleiro Avaliados na TT

No *LS-VisionDraughts*, as informações relacionadas a cada tabuleiro do jogo S_i, avaliado pela rotina de busca *Alfa-Beta*, são armazenados na TT de acordo com a estrutura geral *entry* mostrada abaixo:

```
struct TranspTable{
INT64   hashvalue = v1;
FLOAT   prediction = v2;
MOVE    best_move = v3;
INT     depth = v4;
STRING  scoretype = v5;
```

```
INT     checksum = v6;
}
```

onde *v1* e *v6* representam, respectivamente, as duas *chaves hash* calculada para o estado S_i: *hashvalue* de 64 bits e *checksum* de 32 bits; *v2* e *v3* são, respectivamente, a predição para S_i e o melhor movimento para esse estado, calculado pelo algoritmo de busca; *v4* indica a profundidade da busca que foi calculada a predição *v2*; e, finalmente, *v5* informa se *v2* corresponde ao valor exato da predição de S_i (nesse caso, *v5* é igual a *Exact*) ou um limite superior para esse valor (nesse caso, *v5* é igual a *AtMost*) ou ainda, um limite inferior para esse valor (nesse caso, *v5* é igual a *AtLeast*).

Considerando a estratégia de poda do algoritmo *Alfa-Beta* apresentado na seção 4.5.1, o valor *v5* para o estado S_i é definido da seguinte forma:

- **Exact:** *v5* é *Exact* sempre que não ocorrer qualquer poda durante o cálculo da predição *v2* de S_i;
- **AtMost:** *v5* é *AtMost* sempre que uma poda alfa ocorrer durante o cálculo da predição *v2* de S_i, isto é, S_i é um nó minimizador;
- **AtLeast:** *v5* é *AtLeast* sempre que uma poda beta ocorrer durante o cálculo da predição *v2* de S_i, isto é, S_i é um nó maximizador.

Por exemplo, na estrutura *entry* da TT para o estado de tabuleiro raiz S_1 da figura 18, as duas *chaves hash v1* e *v6* são calculados conforme explicado na seção 4.5.2 e os valores *v2*, *v3*, *v4* e *v5* são respectivamente: *0.4*; "*AÇÃO A*"; *2* e *AtLeast*.

4.5.5 Recuperação de Informações na TT

Sempre que o algoritmo de busca recebe um estado de tabuleiro S_i para ser explorado, ele executa, para cada um de seus filhos C, o seguinte método:

retrieve(C, besteval, bestmove, d, nodeType),

onde C representa o estado de tabuleiro do jogo que está sendo procurado na TT; d representa a profundidade de busca associada a C; *nodeType* indica se o estado pai de C, isto é, S_i, é um nó minimizador ou maximizador; *besteval* e *bestmove* são parâmetros de saída que indicarão, caso ocorra sucesso no procedimento de recuperação do estado C na TT, a predição e a melhor ação associadas ao estado C, respectivamente.

Sempre que esse método é executado, ele primeiro verifica se as informações *besteval* e *bestmove* relacionadas ao estado C estão disponíveis na TT – procedimento conhecido como *teste de ocorrência*. Se esse teste for bem-sucedido, o método verifica se essas

informações satisfazem às *restrições de uso* definida mais adiante nesta seção. Essas restrições devem ser satisfeitas a fim de garantir que o algoritmo *Alfa-Beta* combinado com TT sempre produz, para qualquer estado de tabuleiro arbitrário, o mesmo valor de predição que seria retornado por um algoritmo *Minimax*. Se ambos testes forem bem sucedidos, então o algoritmo *Alfa-Beta*, em vez de avaliar C, simplesmente recupera da TT os valores *v2* e *v3*, apresentados na seção 4.5.4, correspondente à predição e o melhor movimento a ser executado em C. Em seguida, o método instancia o valor *v2* à variável de saída *besteval* e o valor *v3* à variável de saída *bestmove*, retornando-os ao algoritmo *Alfa-Beta*. Se, por outro lado, um dos testes realizados pelo método no início da sua execução (ou ambos) falharem, o algoritmo de busca avalia o nó corrente C, através de chamada recursiva do próprio algoritmo *Alfa-Beta*, com o objetivo de calcular os valores das variáveis *besteval* e *bestmove* e retorná-los para o algoritmo de busca.

Assim, sempre que o *teste de ocorrência* for bem sucedido durante a execução do método *retrieve*, as seguintes restrições (*restrições de uso*) devem ser satisfeitas a fim de garantir que os valores da TT possam ser usados [37]:

1. $depth \geq d$ é a primeira restrição a ser sempre respeitada para permitir ser usado os valores *besteval* e *bestmove* de qualquer nó C disponível na TT – *depth* é o valor $v4$ que indica a profundidade de C na TT, como visto na seção anterior. Essa restrição está vinculada ao fato de que quando mais profundo os valores *besteval* e *bestmove* são calculados, para um determinado estado de tabuleiro, mais preciso eles tornam-se;

2. Se $depth \geq d$ e C é um nó minimizador, seu valor de predição *v2* na TT pode ser usado se seu *scoretype* (ou *v5*) é *Exact*. Caso contrário, isto é, seu *scoretype* é *AtMost* – é importante destacar que o *scoretype* de um nó minimizador pode ser apenas *Exact* ou *AtMost* –, ele só pode ser usado se $v2 \leq \alpha$, onde α é o valor alfa da janela de busca corrente;

3. Se $depth \geq d$ e C é um nó maximizador, seu valor de predição *v2* na TT pode ser usado se seu *scoretype* (ou *v5*) é *Exact*. Caso contrário, isto é, seu *scoretype* é *AtLeast* – é importante destacar que o *scoretype* de um nó maximizador pode ser apenas *Exact* ou *AtLeast* –, ele só pode ser usado se $v2 \geq \beta$, onde β é o valor beta da janela de busca corrente.

4.5.6 Como o algoritmo de busca do LS-VisionDraughts usa *Aprofundamento Iterativo*

A qualidade de um agente jogador que utiliza o algoritmo de busca *Alfa-Beta* está relacionada ao nível de profundidade que ele consegue atingir na árvore do jogo durante a busca pelo melhor movimento. A profundidade da busca está relacionada ao quanto o

jogador consegue "olhar para frente" (do inglês *look-ahead*) e prever as jogadas do adversário. No jogo de Damas, *look-ahead* do agente pode ser restringido devido às limitações impostas pelos recursos computacionais ou ainda, pelo limite de tempo que o agente tem para executar um movimento. A maioria dos jogadores automáticos de Damas utilizam mecanismos para delimitar o tempo máximo permitido de busca. Como o algoritmo *Alfa-Beta* realiza uma busca com profundidade fixa, não existe garantia de que a busca irá completar-se antes que o tempo máximo de busca esgote. Para evitar que o tempo esgote e o algoritmo não consiga escolher o melhor movimento, buscas com profundidade fixa devem ser evitadas. Nesse sentido, a combinação do *Alfa-Beta* com a técnica de *aprofundamento iterativo* garante que o algoritmo retorne um movimento antes que o tempo esgote e ainda, caso haja tempo, permite que o algoritmo busque movimentos com *look-ahead* mais profundos. Tal técnica foi proposta por Larry Atkin no início dos anos 70 com o objetivo de propor uma série de alternativas para controlar o crescimento exponencial de uma busca em árvore [106].

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. Particularmente, *LS-VisionDraughts* executa buscas iterativas de profundidade 2 que são limitadas aos critérios de restrição de profundidade máxima (parâmetro *max-depth*) e intervalo de tempo máximo (parâmetro *max-time*). Inicialmente, o algoritmo *Alfa-Beta* é chamado com profundidade 4, depois com profundidade 6 e assim, sucessivamente, até que o tempo máximo de busca esgote em uma profundidade qualquer $depth = \mathrm{d}$, tal que $4 \leq d \leq max\text{-}depth$. A desvantagem dessa técnica é o processamento repetido de estados de níveis mais rasos da árvore de busca. Dessa forma, a combinação do ID com a TT faz-se necessário, uma vez que acelera o processo de busca iterativa. Os estados de tabuleiros avaliados em níveis mais rasos do ID são recuperados da TT para a iteração do algoritmo em níveis mais profundos.

Outro benefício da combinação do *aprofundamento iterativo* com o *Alfa-Beta* e a TT, é a ordenação parcial da árvore de busca em que coloca-se no ramo mais a esquerda da árvore, o nó filho que obtiver melhor predição obtida na iteração anterior. Assumindo que uma busca mais rasa é uma boa aproximação para outra mais profunda, a melhor ação para um estado S_i na profundidade d será, possivelmente, a melhor ação para o estado S_i na profundidade $d+1$ [72] e, com a ordenação da árvore colocando o melhor movimento no primeiro ramo da árvore de busca, isto é, o ramo mais a esquerda, o melhor movimento deverá ser encontrado mais rapidamente. Por exemplo, a figura 21 mostra o resultado de duas iterações sucessivas do algoritmo *Alfa-Beta*. Veja que a iteração com profundidade d retornou como melhor predição o valor 0.20 referente ao nó B como melhor movimento a ser realizado a partir da raiz. Assumindo que o resultado obtido pela iteração d contém uma boa aproximação do melhor movimento a ser realizado na iteração $d+1$, a árvore de busca é ordenada de forma que o nó filho B fique mais à esquerda da mesma. No exemplo,

após a execução da iteração *d+1*, o nó filho *B* mostrou-se, realmente, a melhor opção de movimento com uma predição 0.30.

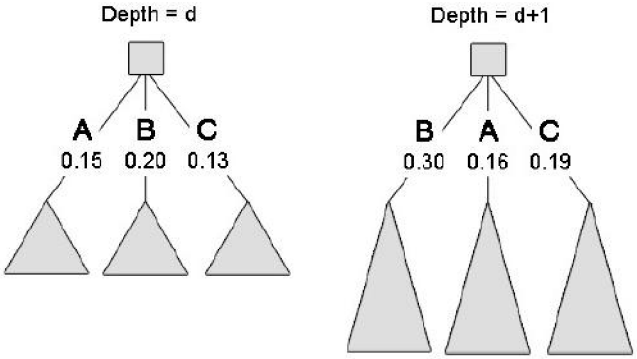

Figura 21 – Exemplo de ordenação da árvore de busca em cada iteração do *aprofundamento iterativo*.

Com a ordenação parcial em cada nível da árvore de busca, o movimento com maior probabilidade de ser o melhor ocupará sempre a primeira posição da árvore, sendo, portanto, o primeiro a ser explorado pelo algoritmo *Alfa-Beta*. Isso faz com que a busca fique ainda mais ágil, já que aumenta as chances de poda do algoritmo, pois os possíveis melhores movimentos são os primeiros a serem explorados. Tal estratégia torna o processo de busca mais eficiente, graças ao aumento do número de podas. Com isso, o *aprofundamento iterativo* consegue explorar níveis mais profundos da árvore de busca na tentativa de encontrar melhores movimentos, aumentando, portanto, o *look-ahead* do jogador.

4.5.7 Incluindo Bases de Final de Jogo no Processo de Busca do LS-VisionDraughts

Como visto nas seções 3.3.1.2 e 3.3.1.3, a adição da base de fim de jogo do *Chinook* dentro da versão estendida do *LS-Draughts* e dentro do *VisionDraughts*, versões predecessoras do *LS-VisionDraughts*, contribuíram, de fato, para melhorar o desempenho geral de tais agentes, além de reduzir a ocorrência de *loops* de final de jogo. Opcionalmente, o algoritmo de busca do *LS-VisionDraughts* também pode ser conectado às bases de fim de jogo do *Chinook* – aqui denominado por *endgame DataBase* (DB). Essas bases foram incluídas com objetivo de antecipar informações relativas à vitória, derrota ou empate para os seguintes estados de tabuleiro, envolvendo até 8 peças:

1. Todos os estados do tabuleiro com 6 ou menos peças;

2. Todos os estados do tabuleiro formados pela combinação de 4 peças x 3 peças;

3. Todos os estados do tabuleiro formados pela combinação de 4 peças x 4 peças.

Detalhes sobre a implementação da biblioteca que dá acesso às bases de dados do *Chinook*, veja [76].

4.6 Experimentos e Análise dos Resultados

Esta seção avalia o desempenho do sistema *LS-VisionDraughts* em relação a seis tipos de análises. Seção 4.6.1 mostra o melhor indivíduo que foi obtido ao longo da evolução do *AG*. Na seção 4.6.2 a ocorrência de *loops* de final de jogo é observada com as versões do *LS-VisionDraughts* treinada com e sem *DB*. Seção 4.6.3 investiga o desempenho do agente em termos de tempo de busca, tempo de treinamento e profundidade de busca (*look-ahead*). A seção 4.6.4 mostra o desempenho do *LS-VisionDraughts* em torneios contra os agentes predecessores, isto é, *NeuroDraughts*, *LS-Draughts* e *VisionDraughts*. Apesar da intenção do presente trabalho em tentar incluir nesses torneios alguns dos agentes não supervisionados descritos nas seções 3.3.1.4 e 3.3.1.5, não foi possível realizar tal tarefa porque, de acordo com os próprios autores desses agentes, não existe interface dos mesmos disponíveis para testes [27]. Seção 4.6.5 tem como objetivo estimar a taxa média de coincidência entre os 15 primeiros movimentos escolhidos pelos agentes *LS-VisionDraughts*, *NeuroDraughts*, *LS-Draughts* e *VisionDraughts* em jogos reais entre eles e os movimentos que, nas mesmas situações, seriam indicados pelo "*conselheiro de movimento*" do forte agente supervisionado de Damas *Cake* – disponível através da plataforma *CheckerBoard* [36]. Finalmente, seção 4.6.6 usa o método de estatística não-paramétrica *Wilcoxon* [107] para validar estatisticamente o ganho real obtido pelo sistema *LS-VisionDraughts* nos testes realizados nas seções 4.6.4 e 4.6.5.

4.6.1 Melhor Indivíduo do AG

A figura 22 mostra os resultados obtidos com a evolução dos 40 indivíduos do *LS-VisionDraughts* ao longo de 30 gerações. Na figura, o *fitness* do melhor indivíduo é comparado com a média de *fitness* da população, a cada grupo de 5 gerações.

A estrutura cromossômica do melhor indivíduo da geração 29, isto é, g_{29}, é apresentada na tabela 8. Ela é composta por 9 *features* ativas do mapeamento *NET-FEATUREMAP*, as quais totalizam 28 bits representando o tabuleiro do jogo de Damas na camada da rede *MLP*. Esse é, portanto, o indivíduo que representará o sistema *LS-VisionDraughts* nas próximas seções de resultados. Em sua versão original, esse indivíduo foi evoluído sem as bases de final de jogo do *Chinook*. Entretanto, com o objetivo de avaliar o uso de *DB* para evitar ocorrências de *loops* de final de jogo, conforme descrito na seção 4.5.7, foi criado uma versão estendida desse indivíduo treinada com as *DBs* do *Chinook*. Os resultados experimentais obtidos com tal versão são apresentados na próxima seção.

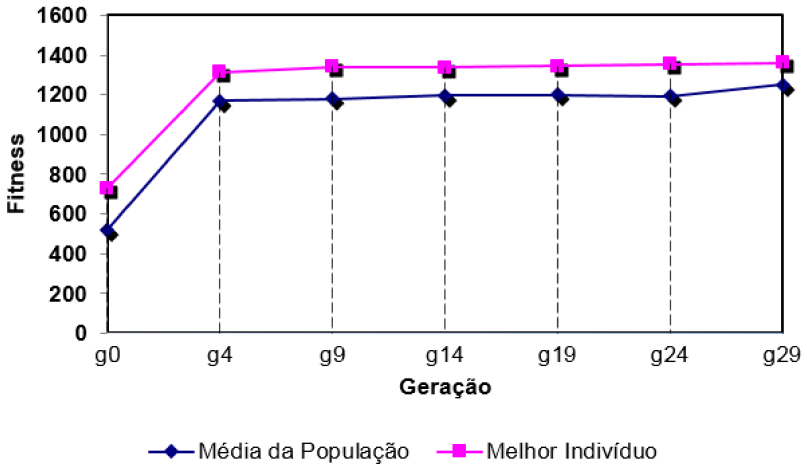

Figura 22 – *Fitness* do melhor indivíduo em relação à média da população.

Tabela 8 – Melhor indivíduo do *LS-VisionDraughts*.

Features	**B**its
F1: PieceAdvantage	4
F2: PieceDisadvantage	4
F3: PieceThreat	3
F4: PieceTake	3
F6: DoubleDiagonal	4
F7: Backrowbridge	1
F8: Centrecontrol	3
F12: KingCentreControl	3
F14: Threat	3

4.6.2 Ocorrência de Loops de Final de Jogo

Esta seção avalia o agente *LS-VisionDraughts* em relação à ocorrência de *loops* de final de jogo. Antes, porém, com o objetivo de simplificar as notações dos agentes automáticos envolvidos nos experimentos, nas tabelas 9 a 14 a serem apresentadas na sequência, os nomes dos agentes *NeuroDraughts*, *LS-Draughts*, *VisionDraughts* e *LS-VisionDraughts* foram simplificados para *Neuro*, *LS*, *Vision* e *LS-Vision*, respectivamente.

A tabela 9 mostra as taxas de ocorrência de *loops* de final de jogo apresentadas nas versões original e estendida (treinada com *DB*) do agente *LS-VisionDraughts* durante seus 1.600 jogos de treinamento. Observe que a inserção da *DB* dentro do processo de treinamento do *LS-VisionDraughts* resultou em uma redução de 66,42% sobre a ocorrência de *loops* de final de jogo.

Tabela 9 – Taxa de ocorrência de *loops* de final de jogo em 1.600 jogos de treinamento.

Jogos com problema de *loop* de final de jogo	
LS-Vision (original)	**LS-Vision (treinado com DB)**
539	181
100%	**33,58%**

4.6.3 Avaliando o Tempo de Busca, o Tempo de Treinamento e o *Look-Ahead*

Esta seção tem como objetivo avaliar o agente *LS-VisionDraughts* em termos de ganho no tempo de busca, tempo de treinamento e aprofundamento na árvore de busca com a adoção do algoritmo *Alfa-Beta* com TT e ID em relação ao algoritmo *Minimax* utilizado pelo agente *LS-Draughts*. A tabela 10 mostra os tempos médios coletados, para ambos algoritmos de busca, e utiliza como referência de 100% o tempo gasto pelo algoritmo *Minimax* para realizar uma busca em profundidade 4. Esse tempo é comparado com o algoritmo *Alfa-Beta* para diferentes valores de profundidade máxima: 4, 6 e 8. Os tempos médios apresentados na tabela 10 foram calculados em um processador Intel Celeron Dual Core E1200 (512k Cache, 1,60 GHz, 800 MHz FSB) com 4 GB de RAM DDR2.

Tabela 10 – a) Tempo médio de busca calculado sobre uma amostra de 10 jogos; b) Tempo médio de treinamento de um indivíduo calculado sobre uma amostra de 10 indivíduos.

Método de Busca (max-depth)	**Tempo Médio de Busca (milissegundos)**	**% sobre o Minimax**
Minimax (4)	49,3	100,0%
Alfa-Beta + TT + ID (4)	9,3	18,8%
Alfa-Beta + TT + ID (6)	37,2	75,4%
Alfa-Beta + TT + ID (8)	173,2	351,2%

(a)

Sistema/Método de Busca (max-depth)	**Tempo Médio de Treinamento (minutos)**	**% sobre o Minimax**
LS :: Minimax (4)	97,5	100,0%
LS-Vision :: Alfa-Beta + TT + ID (4)	17,6	18,0%
LS-Vision :: Alfa-Beta + TT + ID (6)	40,9	41,9%
LS-Vision :: Alfa-Beta + TT + ID (8)	178,6	183,1%

(b)

Observe com base na tabela 10-a, que o *Alfa-Beta* com profundidade máxima 4 e 6 reduziu o tempo médio gasto pelo *Minimax* de 81,2% e 24,6%, respectivamente. Somente na profundidade máxima 8 que o *Alfa-Beta* excedeu o tempo de referência do *Minimax* operando em profundidade 4. Esse mesmo comportamento também pode ser visto na tabela 10-b em relação ao tempo médio de treinamento dos indivíduos para ambas técnicas

de busca. É por isso que este trabalho limitou os parâmetros do aprofundamento iterativo *max-depth* e *max-time* para 8 e 40 milissegundos, respectivamente, a fim de manter um tempo razoável de busca nos *jogos treino* e nos *jogos não treino*. Esses tempos médio de busca também justificam o limite de 30 gerações e 40 indivíduos utilizados no módulo do *AG*, já que esses limites garantem um tempo razoável de treinamento.

4.6.4 Desempenho nos Torneios

Um *torneio de avaliação* foi realizado entre o melhor indivíduo obtido no processo evolutivo do *LS-VisionDraughts* e os agentes baseados em aprendizagem não supervisionada *NeuroDraughts*, *LS-Draughts* e *VisionDraughts* com o objetivo de avaliar o nível de desempenho do novo agente aqui proposto. Para tanto, todos esses agentes automáticos envolvidos no torneio foram treinados em condições similares, isto é:

- Evolução ao longo de 30 gerações, métodos de seleção, *crossover* e mutação conforme descritos na seção 4.2 – técnica adotada apenas pelos agentes *LS-VisionDraughts* e *LS-Draughts*;
- Acesso às bases de fim de jogo (*DB*) para tabuleiros com até 8 peças – parâmetro aplicado somente aos agentes *LS-VisionDraughts*, *LS-Draughts* e *VisionDraughts*;
- Como mostrado na seção 4.4.2, estratégia de treinamento por *self-play* com clonagem onde cada agente joga 4 sessões de 400 jogos de treinamento, sendo metade desses jogos com as peças pretas do tabuleiro e a outra metade com as peças brancas.

Entretanto, em termos de profundidade de busca, todos esses agentes não puderam ser treinados com o mesmo *look-ahead*, já que o baixo desempenho proporcionado pelo algoritmo *Minimax* torna qualquer busca com profundidade maior que 4 inviável (mais detalhes veja seção 4.5). É por isso que *NeuroDraughts* e *LS-Draughts* foram treinados com uma profundidade fixa igual a 4, enquanto que *LS-VisionDraughts* e *VisionDraughts* foram treinados com o aprofundamento iterativo iniciando na profundidade 4 e incrementando 2 níveis, a cada iteração, até uma profundidade máxima 8, limitado ao tempo de busca disponível – esse definido pelo parâmetro *max-time* = 40 milissegundos.

O *torneio de avaliação* consistiu nos seguintes *matches* ou *partidas* (*jogos não treino* citados na seção 4.4):

Grupo I. *Matches* entre agentes treinados sem *DB*:

- LS-VisionDraughts x LS-Draughts;
- LS-VisionDraughts x NeuroDraughts;
- LS-VisionDraughts x VisionDraughts;

Grupo II. *Matches* entre agentes treinados com *DB*:

- ❏ LS-VisionDraughts x LS-Draughts;
- ❏ LS-VisionDraughts x VisionDraughts.

Ambos torneios foram realizados para dois cenários distintos:

1. Todos os jogadores no torneio jogam sem *DB*;
2. Todos os jogadores no torneio jogam com *DB*.

Cada *match* no *torneio de avaliação* consistiu em 28 jogos, onde cada agente joga metade deles como jogador preto e a outra metade como jogador branco. Tal estratégia tem o objetivo explicado na seção 4.4.2. Além disso, com o propósito de garantir uma diversidade de tabuleiros iniciais para os 28 jogos em cada *match* do torneio, foram criados 14 configurações iniciais de tabuleiro, sendo 7 delas geradas pelos movimentos legais iniciais realizados com as peças pretas em um tabuleiro padrão inicial de Damas 8×8 e as outras 7 geradas pelos movimentos legais iniciais realizados com as peças brancas do tabuleiro.

O desempenho do melhor indivíduo do *LS-VisionDraughts* é mostrado na tabela 11.

No cenário 1 da tabela 11, onde todos os jogadores competem em um cenário sem *DB*, *LS-VisionDraughts* mostrou ser bem superior ao agentes *NeuroDraughts* e *LS-Draughts* (esse com as versões treinadas com e sem *DB*), vencendo-os com uma média de 80% vitória. Esse fato mostra que a inclusão de um eficiente método de busca com poda *Alfa-Beta*, combinado com TT e ID, colaborou com perspicácia para melhorar o desempenho do *LS-VisionDraughts* em relação aos seus oponentes que fazem uso do tradicional método de busca *Minimax*. Ainda no cenário 1, os resultados do torneio contra *VisionDraughts* mostram que a taxa de vitória obtida pelo *LS-VisionDraughts* é de aproximadamente 53%. Esse resultado mostra que, embora ambos jogadores adotassem a mesma estratégia de busca, o *AG* foi um fator determinante na seleção de *features*, o que permitiu que *LS-VisionDraughts* superasse *VisionDraughts* dentro do torneio. Jogando com ambas versões do *VisionDraughts*, isto é, versões treinadas com e sem *DB*, *LS-VisionDraughts* não perdeu nenhum jogo.

No cenário 2, onde todos os jogadores competem em um cenário com *DB*, *LS-VisionDraughts* mostrou melhor desempenho com sua versão treinada com *DB*. Esse comportamento mostra que os agentes tendem a ter melhor desempenho em ambientes ou cenários próximos àqueles encontrados em seu treinamento. Por outro lado, o acesso à *DB*, durante os jogos do torneio, permitiram que os jogadores *NeuroDraughts* e *LS-Draughts* melhorassem seus desempenhos em relação aos mesmos jogos do cenário 1.

Em geral, *LS-VisionDraughts* alcançou um alto nível de performance em relação a todos os seus oponentes, obtendo uma taxa média de 70% de vitória, considerando sua pontuação nos cenários 1 e 2.

Tabela 11 – Resultados do torneio de avaliação entre *LS-VisionDraughts* e seus oponentes. a) Cenário 1; b) Cenário 2.

	Cenário 1 *(Todos os jogadores do torneio jogam sem DB)*			
	Matches	**Vitória**	**Empate**	**Derrota**
Grupo I (agentes treinados sem DB)	LS-Vision x Neuro	21	7	0
	LS-Vision x LS	22	6	0
	LS-Vision x Vision	13	15	0
Grupo II (agentes treinados com DB)	LS-Vision x LS	25	2	1
	LS-Vision x Vision	17	11	0
Total		**98**	**41**	**1**
		70%	**29%**	**1%**

(a)

	Cenário 2 *(Todos os jogadores do torneio jogam com DB)*			
	Matches	**Vitória**	**Empate**	**Derrota**
Grupo I (agentes treinados sem DB)	LS-Vision x Neuro	17	11	0
	LS-Vision x LS	18	10	0
	LS-Vision x Vision	19	6	3
Grupo II (agentes treinados com DB)	LS-Vision x LS	21	6	1
	LS-Vision x Vision	24	4	0
Total		**99**	**37**	**4**
		71%	**26%**	**3%**

(b)

4.6.5 Avaliando a Escolha de Movimentos com Relação ao *Cake*

Os testes realizados nesta seção têm como objetivo estimar a taxa média de coincidência entre os 15 primeiros movimentos escolhidos pelos agentes *LS-VisionDraughts*, *NeuroDraughts*, *LS-Draughts* e *VisionDraughts* em jogos reais entre eles e os movimentos que, nas mesmas situações, seriam indicados pelo "*conselheiro de movimento*" do forte e bem sucedido agente supervisionado de Damas *Cake*. Em outras palavras, esses testes visam avaliar o quão perto o "raciocínio" desses agentes está com o "raciocínio" de *Cake*. Assim como estruturado na seção 4.6.4, nos torneios de teste que foram submetidos os agentes, os *matches* foram divididos em dois grupos: o primeiro (I) onde todos os agentes foram treinados sem *DB* e o segundo (II) onde todos os agentes, exceto *NeuroDraughts*, foram treinados com *DB*. Os 14 jogos utilizados para análise foram escolhidos aleatoriamente a partir dos *matches* realizados nos cenários 1 e 2 apresentados na seção 4.6.4. O "*conselheiro de movimento*" utilizado aqui está disponível através da plataforma *CheckerBoard*, que inclui algumas das *engines* (agentes automáticos) atuais de maior sucesso

para Damas [35], [36]. A interface dessa plataforma permite que jogadores humanos e automáticos possam jogar contra as *engines* disponíveis nela.

Este trabalho limitou as análises dos *matches*, jogos realizados entre os agentes na seção 4.6.4), para os 15 primeiros movimentos devido aos seguintes fatos: primeiro, esses *matches* precisaram ser executados manualmente de forma que cada movimento realizado por cada agente pudesse ser comparado com o movimento que seria indicado por *Cake* na mesma situação; segundo, os movimentos iniciais são fundamentais para definir uma vitória ou derrota de um jogador de Damas [35]. A tabela 12 mostra os resultados obtidos nesses testes. Conforme pode ser visto na tabela, *LS-VisionDraughts* supera seus oponentes com uma taxa média de coincidência de movimentos, que seriam executados pelo "*conselheiro de movimento*" de *Cake*, em mais de 60%.

Tabela 12 – Taxa média de coincidência entre os movimentos escolhidos pelo *LS-VisionDraughts* e seus oponentes, quando comparados com àqueles que seriam escolhidos pelo *Cake* na mesma situação.

	Matches (15 movimentos iniciais sobre 14 jogos)	**LS-Vision** (Média de coincidência com *Cake*)	**Oponente** (Média de coincidência com *Cake*)
Grupo I (agentes treinados sem DB)	LS-Vision x Neuro	63.8%	51.4%
	LS-Vision x LS	66,2%	55.2%
	LS-Vision x Vision	61.4%	48.6%
Grupo II (agentes treinados com DB)	LS-Vision x LS	60.5%	40.5%
	LS-Vision x Vision	61.9%	52.8%

4.6.6 Análises Estatística de Wilcoxon

Comparações pareadas são o tipo mais simples de testes estatísticos utilizados para comparar o desempenho de dois ou mais algoritmos quando aplicados a um mesmo problema. Em caso de múltiplos problemas, a comparação requer a definição de um valor para cada par algoritmo/problema. Normalmente, esses valores correspondem a um valor médio que é obtido através de várias execuções [107]. Um dos métodos mais robustos baseado em comparação pareada é o método estatístico chamado *Wilcoxon*, que também é conhecido como teste de ordenação de sinais (do inglês *signed-ranks test*). O objetivo de aplicar o teste de *Wilcoxon* é comparar o desempenho de cada algoritmo, ou pares de algoritmos, como por exemplo A e B, no sentido de verificar se existem diferenças significativas entre os seus resultados, amostras ou populações.

Resumidamente, o teste de *Wilcoxon* opera da seguinte forma. Os resultados obtidos por A são subtraídos dos resultados obtidos por B e a diferença resultante (d) é atribuído o sinal mais (+) ou, caso seja negativa, o sinal menos (-). Essas diferenças são ordenadas

em função de sua grandeza, independentemente do sinal positivo ou negativo. O ordenamento assim obtido é depois apresentado separadamente para os resultados positivos (R^+) de A e negativos (R^-) de B. Os resultados positivos representam momentos em que o algoritmo A supera B e vice-versa para os resultados negativos. O menor dos valores desse segundo grupo, dá-lhe o valor de uma estatística designada por *p-value*, que pode ser consultada na tabela de significância apropriada [107]. A ideia é que se existirem apenas diferenças aleatórias, tal como é postulado pela *hipótese nula*, aqui denominada por H_0, então haverá aproximadamente o mesmo número de ordens elevadas e de ordens inferiores tanto para as diferenças positivas (R^+) quanto para as negativas (R^-). A *hipótese nula* é uma declaração sem efeito ou nenhuma diferença e é esperado ser rejeitada pelo experimentador. Um exemplo de *hipótese nula* é que duas amostras (ou resultados) representam a mesma população, ou seja, não há nenhuma diferença. Se for verificado uma preponderância de baixos resultados para um dos lados (A ou B), isso significa que há existência de muitos resultados elevados para o outro lado, indicando uma diferença em favor de um dos algoritmos. Tal fato contraria a *hipótese nula* H_0. Portanto, dado que a estatística *p-value* reflete o menor total de ordens (R^+ ou R^-), quanto menor for *p-value* mais significativas são as diferenças nas ordenações entre os dois algoritmos. Para mais detalhes sobre o método estatístico de *Wilcoxon*, veja [107].

Particularmente neste trabalho, as *amostras* correspondem aos resultados alcançados pelos *algoritmos* (agentes automáticos) que são candidatos para resolver um determinado *problema* (jogo de Damas). Em outras palavras, o objetivo de usar *Wilcoxon* é verificar se o agente proposto neste trabalho, *LS-VisionDraughts*, apresenta ou não um desempenho diferenciado em relação aos seus adversários. Então, nas análises dos resultados das seções 4.6.4 e 4.6.5, o teste de *Wilcoxon* assume, como ponto de partida, a seguinte *hipótese nula* H_0: *LS-VisionDraughts* possui o mesmo nível de desempenho de seus oponentes.

De acordo com o teste de *Wilcoxon*, os seguintes passos devem ser seguidos a fim de provar que H_0 deve ser rejeitada [27]:

Passo I: Seja T a menor soma dos *ranks* (ordenações) positivas e negativas dentre as diferenças dos pares de amostras (resultados apresentados nas seções 4.6.4 e 4.6.5), isto é, $T = Min\{R^+, R^-\}$. Particularmente neste trabalho, R^+ representa a soma dos *ranks* em que *LS-VisionDraughts* supera seus oponentes, enquanto R^- é a soma dos ranks onde *LS-VisionDraughts* é superado pelos seus oponentes. Use uma tabela de estatística apropriada ou ferramenta para determinar o *teste estatístico*, *valor crítico* ou *p-value* (dado provido pelo teste);

Passo II: Rejeita a *hipótese nula* H_0 se o *teste estatístico* $\leq$ *valor crítico* ou se *p-value* $\leq$ α (nível de significância). Quanto mais baixo é o valor de α, mais forte é a evidência contra a *hipótese nula* H_0.

Dessa forma, as tabelas 13 e 14 mostram os testes estatísticos de *Wilcoxon* correspondentes aos resultados obtidos na quarta e quinta análises apresentadas nas seções 4.6.4 e 4.6.5, respectivamente. Tabela 13 mostra os dados estatísticos de paridade extraídos a partir do desempenho obtido pelo *LS-VisionDraughts* e seus oponentes em 56 jogos de torneio. Tais jogos representam a unificação dos cenários 1 e 2 apresentados na seção 4.6.4, onde a seguinte pontuação por jogo foi definida: 2 pontos para vitória, 1 ponto para empate e 0 ponto para derrota. Os valores R^+, R^- e *p-value* foram computados utilizando o *software* de estatística *SPSS*.

Tabela 13 – Teste da ordenação dos sinais de *Wilcoxon* aplicados aos resultados da seção 4.6.4.

	Cenário 1 + Cenário 2			
	Matches (comparações pareadas em 56 jogos)	**R⁺**	**R⁻**	***p-value***
Grupo I (agentes treinados sem DB)	LS-Vision x Neuro	741	0	7,07E-10
	LS-Vision x LS	820	0	2,54E-10
	LS-Vision x Vision	576	54	9,49E-7
Grupo II (agentes treinados com DB)	LS-Vision x LS	1127	49	2,14E-10
	LS-Vision x Vision	861	0	1,52E-10

Conforme pode ser visto na tabela 13, em todas as situações, *LS-VisionDraughts* é superior aos agentes *NeuroDraughts*, *LS-Draughts* e *VisionDraughts* com um alto nível de significância $\alpha = 0{,}01$. Tal fato representa que há 1% de chance da *hipótese nula* H_0, isto é, equivalência de desempenho entre *LS-VisionDraughts* e seus oponentes, ocorrer na base (amostra) analisada. Em outras palavras, os dados apresentados na tabela 13 rejeitam fortemente H_0, o que indica que os resultados alcançados pelo *LS-VisionDraughts*, nos jogos de torneio realizado na seção 4.6.4, são realmente diferenciados em relação àqueles obtidos pelas versões predecessoras. Analogamente, a tabela 14 mostra os valores R^+, R^- e *p-value* computados para todas as comparações pareadas que refletem a taxa média de coincidência entre os movimentos escolhidos pelo *LS-VisionDraughts* e seus oponentes, quando comparados com aqueles que seriam escolhidos pelo *Cake* na mesma situação. Essas taxas médias de coincidência foram obtidas nos torneios executados na seção 4.6.5.

Conforme apresentado na tabela 14, nos torneios relacionados ao grupo I, *LS-VisionDraughts* mostrou ser superior aos agentes *NeuroDraughts* e *VisionDraughts* com um nível de significância $\alpha = 0{,}01$ e superior ao *LS-Draughts* com um nível de significância $\alpha = 0{,}02$. Já com relação ao jogos do grupo II, *LS-VisionDraughts* supera os agentes *LS-Draughts* com um nível de significância $\alpha = 0{,}01$ e *VisionDraughts* com um nível de significância $\alpha = 0{,}04$. Tais fatos rejeitam fortemente a *hipótese nula*, o que indica que os 15 movimentos iniciais executados pelo *LS-VisionDraughts* são bem mais próximos daqueles indicados por *Cake*, quando comparados com os movimentos executados pelos seus oponentes.

Tabela 14 – Teste da ordenação dos sinais de *Wilcoxon* aplicados aos resultados da seção 4.6.5.

	Matches (comparações pareadas em 14 jogos)	R^+	R^-	*p-value*
Grupo I (agentes treinados sem DB)	LS-Vision x Neuro	78,0	0,0	0,00204
	LS-Vision x LS	79,5	11,5	0,01698
	LS-Vision x Vision	63,5	2,5	0,00645
Grupo II (agentes treinados com DB)	LS-Vision x LS	87,0	4,0	0,00357
	LS-Vision x Vision	57,0	9,0	0,03234

4.7 Considerações Relativas ao Capítulo

Este capítulo apresentou como a combinação de *AGs* com um eficiente módulo de busca baseado em algoritmo *Alfa-Beta* com tabela de transposição, aprofundamento iterativo e ordenação parcial da árvore de busca, pode melhorar o processo de aprendizagem de um agente automático que aprende por *AR*. Os bons resultados obtidos pelo agente proposto neste capítulo, *LS-VisionDraughts*, contra suas versões predecessoras, foram obtidos devido a dois fatores principais: primeiro, a automatização das *features* que representam os tabuleiro de Damas na entrada da rede *MLP* e segundo, devido a melhoria na estratégia de busca na árvore do jogo pelo melhor movimento. Além disso, o uso de bases de final de jogo do *Chinook* provou ser uma ferramenta útil para reduzir a ocorrência de *loops* de jogadas executadas pelo agente no fim dos jogos.

É importante ressaltar aqui que apesar do sistema *LS-VisionDraughts* ser comprovadamente eficaz em suas tomadas de decisão, quando comparado com os agentes que o precederam, tal arquitetura apresenta o inconveniente de ser extremamente previsível, executando sempre o mesmo movimento diante de um mesmo tabuleiro e independente do adversário. Note que tal comportamento não permite ao agente *LS-VisionDraughts* evoluir seu nível de jogo, evitando alcançar tabuleiros de jogos desfavoráveis (que o leva a derrota), quando enfrenta adversários mais difíceis. Em outras palavras, *LS-VisionDraughts* não é capaz de evoluir seu nível de jogo observando sua própria experiência contra diferentes adversários. Neste sentido, com o objetivo de introduzir uma abordagem não determinística de tomada de decisão, o presente trabalho propõe, na sequência, a construção da arquitetura híbrida *ACE-RL-Checkers* que será detalhada no capítulo 5.

Capítulo 5

ACE-RL-Checkers

Este capítulo apresenta o sistema *ACE-RL-Checkers*, uma nova abordagem híbrida não determinística que combina as técnicas de *AM Aprendizagem por Reforço* e *Elicitação Automática de Casos*, conforme objetivos 3 e 4 traçados na seção 1.3.1. A principal motivação deste trabalho para a construção do *ACE-RL-Checkers* é fundamentada na potencialidade da abordagem híbrida proposta por De Jong em [93] e nos resultados obtidos por Powell com a versão probabilística da técnica *EAC* em [21], conforme arquiteturas apresentadas nas seções 3.3.2.1 e 3.4.1.

A abordagem híbrida de De Jong propõe que a aplicação adequada de um algoritmo *EBL* em combinação com um *solucionador de problemas* (agente) estático pode conduzir ao desenvolvimento de um sistema híbrido muito melhor que esse último e com grande capacidade de recuperar e aplicar rapidamente as ações executadas no passado e que estão armazenadas em uma *base de conhecimento*. De fato, os resultados obtidos por De Jong em [93] comprovam a superioridade de tal arquitetura híbrida em relação às diversas arquiteturas estáticas testadas, as quais não fazem uso de um algoritmo *EBL*, isto é, utilizam apenas agentes estáticos. Entretanto, por outro lado, Powell demonstrou em [21] que a escolha inadequada de um algoritmo *EBL* pode comprometer o desempenho geral de um sistema híbrido, tal como aquele proposto por De Jong em [93]. Mais especificamente, em [21], Powell observou que a superioridade de *CHEBR*, sistema *EBL* com abordagem probabilística, em relação a *GINA*, sistema híbrido que combina uma versão não probalilística da técnica *EBL* com o conhecimento provido por um agente estático, foi alcançada devido ao poder da exploração pseudo-aleatória inerente ao algoritmo *EBL* adotado pelo primeiro sistema. Em outras palavras, o fato de *GINA* ter adotado um *SRBC* não probabilístico recaiu, a longo prazo, no mesmo problema encontrado na arquitetura do sistema *LS-VisionDraughts*, isto é, tais sistemas apresentam o inconveniente de serem extremamente previsíveis, executando sempre o mesmo movimento diante de um mesmo tabuleiro e independente do adversário. Já o algoritmo *EBL* adotado por Powell em [21] é um *SRBC* probabilístico que apresenta um comportamento extremamente adaptativo e com habilidades para explorar pseudo-aleatoriamente o espaço de busca. Fato que per-

mitiu que *CHEBR* superasse a versão para damas de *GINA* [21].

Em contrapartida, apesar da eficiência e dos resultados obtidos por Powell com a técnica *EAC* em [21], o presente trabalho identificou um problema no desempenho do agente *CHEBR* ao reproduzí-lo e testá-lo contra o agente *LS-VisionDraughts*. Foi observado, ao longo dos jogos de treinamento, uma alta frequência de tomada de decisão baseada apenas em ações aleatórias – tal problema foi mapeado ao totalizar a quantidade de ações aleatórias sugeridas pela técnica *EAC* em relação a quantidade total de movimentos executados pelo agente nos jogos. Esse problema torna-se ainda mais acentuado quando a quantidade de casos disponíveis na *biblioteca* da técnica *EAC* é extremamente baixa em função do pouco conhecimento que o agente possui em relação ao perfil de jogadas de seu adversário. Nesses casos onde não há nenhum conhecimento armazenado na *base de conhecimentos*, o agente simplesmente executa movimentos pseudo-aleatórios tentando explorar novas regiões do espaço de busca. A reprodução do agente *CHEBR* de Powell foi conduzida no sentido de cumprir o segundo objetivo proposto na seção 1.3.1: avaliar o desempenho geral da arquitetura dinâmica de Powell em jogos de torneio contra a eficiente arquitetura estática *LS-VisionDraughts*. Maiores detalhes desses experimentos são apresentados na seção 5.7.

Considerando os fatos mencionados acima, o sistema híbrido *ACE-RL-Checkers* propõe a seguinte combinação de técnicas de *AM*: utilizar a versão probabilística da técnica *EAC* de Powell [21] como ferramenta *EBL* e o conhecimento da *MLP* do melhor indivíduo do *LS-VisionDraughts* como *solucionador de problemas* e alicerce para a geração da *base de conhecimentos* da técnica *EAC*. Mais especificamente, o objetivo é utilizar o conhecimento provido por um agente estático treinado por *AR* para direcionar a exploração aleatória da técnica *EAC* para regiões mais promissoras no espaço de busca, fato que refina a qualidade das tomadas de decisão dinâmica do agente, uma vez que reduz a quantidade de execução de ações aleatórias nos jogos e consequentemente, reduz a quantidade total de casos armazenados na *base de conhecimentos*. Por outro lado, a nova dinâmica aleatória do *ACE-RL-Checkers*, proporcionada pelo módulo de seleção pseudo-aleatória de casos e norteada por conhecimento, introduz adaptabilidade ao agente, uma vez que as tomadas de decisão não serão mais determinísticas e as escolhas de movimento do agente serão baseadas na dinâmica corrente de cada jogo.

As próximas seções deste capítulo são organizadas da seguinte forma: seção 5.1 apresenta a arquitetura geral do *ACE-RL-Checkers*; seção 5.2 mostra como um caso é representado e armazenado na TT do *SRBC*; seção 5.3 mostra como é tratado o problema de colisão da TT do *SRBC*; seção 5.4 mostra como os casos são recuperados da biblioteca do agente; seção 5.5 mostra as três estratégias adotadas por esta pesquisa para atualização do valor do *rating* dos casos: estratégia original de Powell proposta em [21] e as duas novas estratégias propostas no quarto objetivo da seção 1.3.1; na sequência, seção 5.6 mostra o ciclo *SRBC* do sistema híbrido *ACE-RL-Checkers* aqui proposto e o que desse

ciclo difere da arquitetura original do sistema *CHEBR* de Powell proposto em [21]; por fim, seção 5.7 apresenta os resultados obtidos em torneio com a nova abordagem híbrida de *AM*, o agente *ACE-RL-Checkers*.

5.1 Arquitetura Geral do *ACE-RL-Checkers*

A figura 23 mostra a arquitetura geral do sistema *ACE-RL-Checkers* tanto para os *jogos treino* (sessões de treinamento) quanto para os *jogos não treino* (torneio), a qual o processo de aprendizagem é sempre contínuo e *online*. Resumidamente, sempre que o agente precisa escolher um novo movimento, aqui denominado por a_{t+1}, pois $t + 1$ representa um estado futuro ao tempo atual t, o fluxo do processo mostrado através dos passos 1 a 9 da figura 23 é sempre acionado. Primeiramente, o estado de tabuleiro corrente, aqui denominado por S_t, é apresentado ao *Módulo EAC* #1 (passo 1 na figura). Esse módulo utiliza a versão probabilística da técnica *EAC* proposta por Powell [21] para escolher qual o movimento a_{t+1} mais adequado a ser executado em S_t. Nesse sentido, o *Módulo EAC* tenta recuperar todos os casos (pares estado/ação) que já foram executados no passado pelo agente, no estado S_t, que estão armazenados na *biblioteca* de casos do *SRBC*, #2. Se a tentativa do passo #2 falhar, isto é, o estado S_t é um novo estado para o agente e, portanto, não existe nenhum caso aplicável a S_t armazenado na *biblioteca* de casos, então o *módulo AR* é acionado, #3. Esse módulo corresponde a *MLP* do melhor indivíduo do *LS-VisionDraughts* e é responsável por escolher a melhor ação a ser executada em S_t utilizando a estratégia de busca em árvore *Alfa-Beta* com TT e ID descrita na seção 4.5. A melhor ação a_{t+1} sugerida pelo *módulo AR* é então retornada para o *Módulo EAC*, #4. Tal ação é executada sobre o tabuleiro S_t, #5, produzindo o estado S_{t+1}, #6, e o caso C_{t+1} formado pelo novo par (S_t, a_{t+1}) é armazenado na *biblioteca* de casos do *SRBC*, #7, sem nenhum valor de *rating* definido (detalhes sobre o parâmetro *rating* são apresentados nas seções 5.2 e 5.5). Tal valor só é atualizado após o fim de cada partida, isto é, após o estado *fim-de-episódio*, que é quando o desempenho final do agente no jogo é conhecido: sucesso (vitória) ou fracasso (empate ou derrota), passos #8 e #9 da figura. Se, entretanto, por outro lado, a tentativa do passo #2 obter sucesso, isto é, existem casos armazenados aplicáveis a S_t armazenados na *biblioteca* do agente, então os passos #3 e #4 são descartados e o próprio *Módulo EAC* será responsável por escolher pseudo-aleatóriamente a ação a_{t+1} mais adequada a ser executada no passo #5, dentre todos os casos recuperados, produzindo o estado S_{t+1}, #6 – a partir daí, os demais passos da figura 23 continuam os mesmos. O processo de escolha pseudo-aleatória de movimentos do *Módulo EAC* é descrito com detalhes na seção 5.6.1.2 e pode, inclusive, retornar uma nova ação aleatória, produzindo um novo caso, daí a necessidade do passo #7 da figura.

Com relação a arquitetura do *ACE-RL-Checkers* apresentada na figura 23, dois pontos merecem ser destacados: 1) Para cada adversário, a *biblioteca* de casos do *SRBC* é sem-

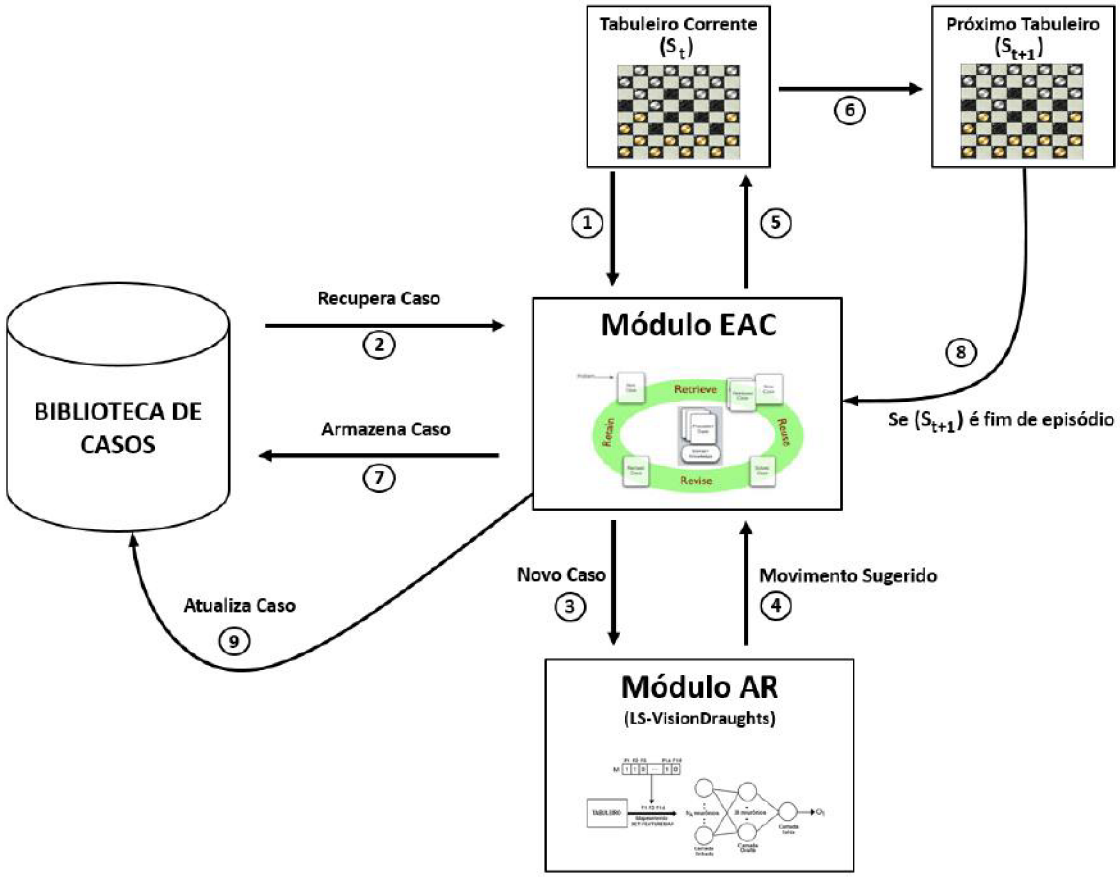

Figura 23 – Arquitetura geral do sistema *ACE-RL-Checkers*.

pre inicializada (zerada), daí a necessidade de *jogos treino* (sessões de treinamento) para que o agente aprenda o perfil de jogadas de seu oponente e consequentemente, refina seu processo de tomada de decisão baseando-se na dinâmica corrente de cada jogo. Assim, ao final dos *jogos treino*, o sistema *ACE-RL-Checkers* terá uma *biblioteca* de casos especializada para cada adversário; 2) Como a arquitetura do *ACE-RL-Checkers* é baseada em uma técnica *EBL*, ou seja, *aprendizagem baseada em experiência*, sua *biblioteca* de casos permanece em constante processo de aprendizagem, até mesmo durante os *jogos não treino* (torneio).

As próximas seções descrevem com detalhes o processo de armazenamento e recuperação de casos da *biblioteca* do agente, a atualização dos valores dos *ratings* dos casos, bem como o ciclo e os módulos *EAC* e *AR* do *SRBC* aqui proposto.

5.2 Representação do Caso e Armazenamento na TT do SRBC

No *ACE-RL-Checkers* cada caso corresponde, logicamente, a uma tupla do conjunto $C = \{(S_1, a_1), ..., (S_1, a_i), ..., (S_2, a_2), ..., (S_n, a_m)\}$, onde o conjunto $\{S_1, S_2, ..., S_n\}$ repre-

senta todos os estados de tabuleiro de fato apresentados ao agente durante os jogos e o conjunto $\{a_1, a_2, ..., a_i, ..., a_m\}$ representa todos os movimentos de fato executados pelo agente para cada estado S_i, onde $i \in \{1, 2, ..., n\}$. Além da tupla (S_i,a_j), onde $S_i \in \{S_1, S_2, ..., S_n\}$ e $a_j \in \{a_1, a_2, ..., a_i, ..., a_m\}$, cada caso possui os seguintes atributos: quantidade de vitórias, empates e derrotas obtidas pelo agente com a execução da ação a_j a partir do estado S_i, quantidade de vezes que o estado S_i apareceu para o agente, quantidade de vezes que a ação a_j foi executada a partir do estado S_i, identificador se a ação a_j foi sugerida pela *MLP* do *LS-VisionDraughts* e o valor do *rating* para o caso (S_i,a_j).

O *rating* de um caso é responsável por indicar ao *SRBC* o quanto um par (S_i,a_j) é bom para o agente baseado em suas experiências de sucesso (vitórias), alcançadas até aquele momento, ao executar a ação a_j em S_i. Em outras palavras, é o *rating* que classifica a qualidade de um caso armazenado na *base de conhecimentos* do agente. Existem diversas formas para atualizar o *rating* de um caso, dentre as quais, podem-se destacar o uso de atributos estatísticos de *performance* (desempenho) obtido pelo agente ao longo dos jogos, como, por exemplo, média de vitórias, quantidade de execuções e outros. A seção 5.5 apresenta com mais detalhes as estratégias adotadas por esse trabalho para a atualização dos valores dos *ratings* dos casos gerados no contexto do sistema *ACE-RL-Checkers*. Como será visto, tais estratégias são de suma importância para aprimorar a acurácia dos casos gerados pelo agente e a consequente conversão da técnica *EBL* adotada.

Por outro lado, fisicamente, cada caso (S_i,a_j) é armazenado em uma *Tabela de Transposição* utilizando a mesma estratégia de definição de *chave hash* descrita na seção 4.5.2 para um determinado tabuleiro de jogo S_i. A estrutura geral *ENTRY_CBR* utilizada em cada entrada na TT do *SRBC* é definida a seguir:

```
struct ENTRY_CBR{
BOARD   board = v1;
INT64   hashvalue = v2;
INT     checksum = v3;
MOVE    movementsB[] = v4;
INT     faced_times_countB = v5;
MOVE    movementsW[] = v6;
INT     faced_times_countW = v7;
}
```

onde *v1* é a representação vetorial de 32 posições para o estado de tabuleiro S_i; *v2* e *v3* representam, respectivamente, as duas *chaves hash* calculada para o estado S_i: *hashvalue* de 64 bits e *checksum* de 32 bits; *v4* e *v6* são, respectivamente, o vetor de movimentos (ações) executadas pelo jogador preto (quando o agente joga com as peças pretas do tabuleiro) e pelo jogador branco (quando o agente joga com as peças brancas do tabuleiro) a partir do estado de tabuleiro S_i; *v5* e *v7* representam, respectivamente, a quantidade

de vezes que o tabuleiro S_i é apresentado ao jogador preto e branco.

A estrutura *MOVE*, utilizada pela estrutura geral *ENTRY_CBR* da TT, representa o movimento a_j da tupla (S_i,a_j) e é definida pela estrutura abaixo:

```
struct MOVE{
INT     from = v8;
INT     to = v9;
INT     exchange = V10;
INT     SeqDIRS[] = v11;
INT     wins_count = v12;
INT     draws_count = v13;
INT     defeats_count = v14;
INT     executed_times_count = v15;
FLOAT   CBRrating = v16;
INT     isMoveSuggByMLP = V17;
}
```

onde *v8* e *v9* representam, respectivamente, as posições de origem e destino do movimento a_j; *v10* e *v11* são, respectivamente, o tipo de movimento, 0 – movimento simples ou 1 – captura de peças do adversário, e o caminho a ser realizado no tabuleiro para capturar as peças do adversário (quando aplicável); *v12*, *v13* e *v14* são, respectivamente, as quantidades de vitórias, empates e derrotas obtidas pelo agente com a execução do movimento a_j; *v15* é a quantidade de vezes que o movimento a_j foi executado pelo agente; *v16* é o valor do *rating* para o caso ou tupla (S_i,a_j); *v17* indica se o movimento a_j foi sugerido pela *MLP* do agente estático *LS-VisionDraughts*.

Por fim, a representação vetorial do estado de tabuleiro S_i, também utilizada pela estrutura geral *ENTRY_CBR* da TT, é definida a seguir:

```
struct BOARD{
INT    p[32] = v18;
};
enum BOARDVALUES {EMPTY = 0, BLACKMAN, WHITEMAN, BLACKKING, WHITEKING};
```

onde *v18* é um vetor de 32 posições correspondente ao tabuleiro padrão de Damas 8×8, sendo que cada quadrado é preenchido por um dos valores do tipo de enumeração *BOARDVALUES*: 0 (vazio), 1 (peça simples preta), 2 (peça simples branca), 3 (dama preta) ou 4 (dama branca).

Observe que no *ACE-RL-Checkers*, cada caso do conjunto $\{(S_i, a_1), (S_i, a_2), ..., (S_i, a_m)\}$, onde $\{a_1, a_2, ..., a_m\}$ representa o conjunto das m ações executadas pelo agente a partir do estado S_i, é logicamente independente um do outro, mas fisicamente é armazenado em uma mesma entrada *ENTRY_CBR* da TT, devido a estratégia de definição de *chave hash* proposta por *Zobrist*, conforme processo descrito na seção 4.5.2.

5.3 Tratamento do Problema de Colisão na TT do SRBC

A TT utilizada pelo *ACE-RL-Checkers*, para armazenamento dos casos em *biblioteca*, também trata os dois tipos de problema de colisão identificados por Zobrist em [102] e descritos na seção 4.5.3: *erro tipo 1* e *erro tipo 2*. Particularmente, em relação ao *erro tipo 2*, onde é utilizado dois níveis de estrutura de endereço na TT, foi implementada uma estratégia de substituição diferente daquela proposta por Breuker em [105], isto é, esquemas de substituição *Deep* e *New*. Tais esquemas são válidos para armazenar posições de tabuleiro do jogo de Damas em conjunto com uma estratégia de busca em árvore, onde profundidade e avaliação (*predição*) são atributos importantes. Daí a necessidade da estratégia *Deep*, pois sempre que um determinado endereço de entrada na TT armazenar informações referentes ao mesmo estado de tabuleiro S_i, em ambos os níveis, isso significa que a informação armazenada no primeiro nível foi obtida a partir de uma posição mais profunda na árvore do jogo, ou seja, informação mais precisa. Esse tipo de cenário não é aplicável para o *SRBC* implementado neste capítulo, o qual requer apenas o uso do esquema de substituição *New*, onde o segundo nível da TT só será utilizado quando ocorrer o problema de colisão de *erro tipo 2*. Nesse caso, um mesmo endereço de entrada na TT só armazenará informações referentes a dois estados diferentes.

5.4 Recuperação de Casos da TT do SRBC

Sempre que o *Módulo EAC* precisa recuperar casos armazenados na *biblioteca* do *SRBC*, aplicáveis a um determinado estado S_i (passo #2 da figura 23 apresentada na seção 5.1), ele executa o seguinte método:

GetEntry(B, Moves[], storedMovesCount, isBlack),

onde B representa o estado de tabuleiro do jogo S_i que está sendo procurado na TT; *Moves[]* e *storedMovesCount* são parâmetros de saída que representam, respectivamente, a lista e a quantidade de ações executadas pelo agente, até aquele momento, no estado S_i – logicamente, cada ação corresponde a um caso com valor de *rating* específico; *isBlack* indica qual lista de movimentos recuperar da TT: movimentos executados pelo jogador preto ou pelo jogador branco.

Observe que o método *GetEntry* utiliza a mesma estratégia de *matching* adotada por Powell em [21], isto é, estratégia *CHEBR-Exact* descrita na seção 3.4.1.

5.5 Equação de Atualização do *Rating*

A atualização do valor do atributo *rating* de cada caso, gerado por ações executadas pelo sistema *ACE-RL-Checkers* ao longo dos jogos, só acontece ao final de cada partida que é quando o desempenho final do agente é conhecido: sucesso em caso de vitória ou falha em caso de empate ou derrota. Nesta seção, 3 estratégias para atualização do *rating* são investigadas. A primeira é a estratégia de *Decaimento de Memória* – do inglês *Decaying Memory* (DM) – proposta por Powell, em [21], na versão probabilística da técnica *EAC*. As outras duas estratégias são objetos de pesquisa deste trabalho com o intuito de investigar o quanto a inclusão de novas formas para calcular o valor do *rating* dos casos pode ajudar a melhorar sua acurácia e o consequente desempenho geral do sistema *ACE-RL-Checkers*. Para tanto, duas estratégias distintas são propostas e avaliadas aqui: eliminação do *decaimento de memória* adotado em [21] e a inserção do dilema *exploration/exploitation* inerente à técnica UCT – técnica bastante utilizada com sucesso em jogadores de Go automático [69].

As próximas subseções são estruturas da seguinte forma. Inicialmente, é apresentada a estratégia *DM* adotada pela versão original da técnica *EAC* proposta em [21]. Em seguida, as duas novas estratégias propostas no quarto objetivo da seção 1.3.1 são apresentadas.

5.5.1 Decaimento de Memória

A estratégia *DM* foi proposta originalmente por Powell em [21] com o objetivo de atualizar o valor do *rating* dos casos gerados pela versão probabilística da técnica *EAC*. Tal estratégia é utilizada pelo sistema *ACE-RL-Checkers* e é definida pela seguinte equação:

$$r_n = \begin{cases} \frac{1}{2}s_0 & \text{para } n=0, \\ \frac{1}{2}s_n+\frac{1}{2}r_{n-1}=\left(\frac{1}{2}\right)^1 s_n+\left(\frac{1}{2}\right)^2 s_{n-1}+\ldots+\left(\frac{1}{2}\right)^{n+1} s_0 & \text{para } n>0, \end{cases} \tag{7}$$

onde $r_n \in [0,1]$ e $s_n \in \{0,1\}$ representam, respectivamente, o valor do *rating* e o resultado do jogo – 1 em caso de sucesso (vitória) e 0 em caso de falha (empate ou derrota) – da *n-ésima* aplicação de um caso em um jogo, dentre uma sequência de jogos disputados pelo agente. O propósito dessa fórmula é prover uma estratégia de *decaimento de memória*, em que as consequências da aplicação de um caso no início da vida do sistema, quando a *biblioteca* do *SRBC* é desprovida de conhecimento e casos são aplicados aleatoriamente, são rapidamente esquecidos. Em tal estratégia, apenas as aplicações mais antigas de um caso, não os próprios casos, são esquecidas por redução matemática de seus efeitos sobre o *rating* atual do caso. Observe que o *rating* de um caso tende a "flutuar" inicialmente em torno de 0,5 no "início da vida do agente", uma vez que sucesso e fracasso são igualmente prováveis. Assim, à medida que o agente ganha experiência, o *rating* do caso tende a melhorar, tendendo para 1 (altamente bem sucedido), ou piorar, tendendo para 0 (completamente ineficaz) [21].

5.5.2 Memória Positiva Geral

A primeira estratégia alternativa proposta pelo presente trabalho para substituir o *DM* no *ACE-RL-Checkers* é a estratégia de *Memória Positiva Geral* – do inglês GPM. Essa estratégia foi proposta originalmente por Powell em uma versão não probabilística da técnica *EAC* em [20] e é definida pela seguinte equação:

$$r_n = \frac{c + W}{2c + W + L}, \tag{8}$$

onde c é um valor constante usado para desenfatizar movimentos realizados no "início da vida do agente", que é quando o agente está simplesmente aprendendo movimentos válidos do jogo; W e L são, respectivamente, a quantidade de vitórias e derrotas obtidas pelo agente no momento em que um caso arbitrário é usado nos jogos pela *n-ésima* vez. Observe que a estratégia *GPM* difere da estratégia *DM* em dois pontos: primeiro, jogos que terminam empatados não influenciam no valor do *rating* dos casos; segundo, a equação *GPM* não utiliza a estratégia de *decaimento de memória*, onde os movimentos mais antigos tem bem menos impacto do que os movimentos executados recentemente pelo agente. *GPM* destina-se a simular uma *lembrança positiva geral* que um jogador pode ter em relação a um determinado movimento e as consequências gerais, a longo prazo, derivadas da aplicação repetida do referido movimento [20]. Assim como ocorre na estratégia *DM*, à medida que o agente ganha experiência e joga mais vezes, o *rating* da equação 8 pode tender a 1 (um ótimo caso) ou 0 (um caso totalmente ineficaz).

Vale destacar também que apesar de Powell ter proposto ambas estratégias, *GPM* e *DM*, elas não foram testadas juntas na versão probabilística da técnica *EAC* em [21]. Essa combinação é então investigada pelo presente trabalho com o intuito de avaliar o desempenho de ambas estratégias no contexto de casos gerados pela versão probabilística da técnica *EAC* adotada pelo sistema *ACE-RL-Checkers*. Para isso, a equação *DM* é substituída pela equação 8. O valor para a constante c da equação 8 foi empiricamente definida como zero, o qual proveu os melhores resultados.

5.5.3 Memória de Confiança Superior

A segunda estratégia alternativa proposta para substituir o *DM* no *ACE-RL-Checkers* é a estratégia *Memória de Confiança Superior* – do inglês UCM. Tal estratégia é uma proposta do presente trabalho em adaptar o algoritmo UCT, utilizado com sucesso pela política de seleção de nós da técnica de busca em árvore *Monte Carlo* – do inglês *Monte Carlo Tree Search* (MCTS), dentro do contexto *SRBC*. Para escolher o movimento apropriado, o algoritmo *UCT* busca balancear a qualidade de nós promissores (exploitation) com a exploração de nós da árvore pouco simulados (exploration) para cada movimento candidato [69]. A estratégia *UCM* proposta aqui é definida pela seguinte equação:

$$r_n = Q(S_i, a_j) + C \cdot \sqrt{\frac{\ln N(S_i)}{N(S_i, a_j)}}, \tag{9}$$

onde $Q(S_i, a_j)$ é a quantidade média de vitórias para cada caso (S_i,a_j); $N(S_i, a_j)$ é a quantidade de vezes que o movimento a_j foi executado a partir do tabuleiro de jogo S_i; e $N(S_i)$ é a quantidade de vezes que o tabuleiro de jogo S_i foi apresentado para o agente ao longo dos jogos. C é uma constante escalar que pondera o dilema *exploitation* e *exploration* da equação 9. Observe que cada vez mais que uma ação a_j é executada, o denominador $N(S_i, a_j)$ do segundo termo da equação *UCM* tende a aumentar e consequentemente, diminuir o fator *exploration* da equação 9. Esse fato faz com que outras ações a partir do tabuleiro do jogo S_i, que foram poucas vezes executadas, passam a ter maior prioridade e consequentemente, maior fator *exploration*. Em outras palavras, ter um maior fator *exploration* implica que o agente precisa explorar novas regiões no espaço de busca, as quais podem ser promissoras para o agente, com o objetivo de obter um melhor desempenho no jogo. O mesmo caso aplica-se para o fator *exploitation* (primeiro termo da equação 9), isto é, quando melhor for a média de vitórias das ações executadas a partir do tabuleiro de jogo S_i, maior será a possibilidade dessas ações serem selecionadas no futuro pela equação UCM.

5.6 Ciclo SRBC

O ciclo *SRBC* do sistema híbrido *ACE-RL-Checkers* proposto aqui é composto por 4 passos principais, conforme pseudo-código apresentado no algoritmo 3 através das linhas 3 a 15. Esse ciclo é uma expansão da arquitetura geral apresentada na seção 5.1 e repete-se para cada jogo disputado entre o *ACE-RL-Checkers* e seus adversários, seguindo a dinâmica detalhada abaixo:

1. *Recuperação de Casos*: toda vez que o agente tem de executar um movimento sobre um determinado tabuleiro de jogo corrente S_t (linha 6), a função *MatchingCases()* na linha 7 recupera um conjunto $M \in C$, onde C é a *biblioteca* de casos do *ACE-RL-Checkers*, contendo todos os casos que são aplicáveis para S_t. Mais especificamente, o conjunto M é obtido através do parâmetro de saída *Moves[]* do método *GetEntry()* apresentado na seção 5.4 – tal método é acionado pela própria função *MatchingCases()*. Observe que cada elemento do conjunto M corresponde a um caso ou uma ação diferente aplicável a S_t que já foi executada (experimentada) no passado pelo agente. O conjunto M recuperado pela função *MatchingCases()* é retornado em ordem decrescente do valor do atributo *rating* de seus casos – a importância dessa ordem é melhor explicada na seção 5.6.1.2. É importante destacar que o conjunto M retornado nesse passo pode ser vazio, o que significa que S_t é um novo tabuleiro

de jogo apresentado para o *ACE-RL-Checkers* e que ainda não foi experimentado pelo agente;

2. *Reuso de Casos*: este passo é responsável por definir qual movimento deve ser executado pelo *ACE-RL-Checkers* em S_t em função do conjunto M retornado pelo passo 1 (*recuperação de casos*). Para tanto, a função *MoveDecision()* da linha 9 (função que é detalhada na seção 5.6.1) é acionada. Tal função retorna uma ação A_{t+1} a ser aplicada em S_t;

3. *Revisão de Casos*: este passo é responsável por aplicar a ação A_{t+1} escolhida no passo 2 (*reuso de casos*) através do método *ApplyAction()* na linha 10, produzindo o estado S_{t+1}. Em seguida, tal ação é associada a um caso ou tupla (S_t,A_{t+1}) e então é armazenada em um *array* AC (linha 13) contendo todos os casos gerados por ações executadas pelo agente ao longo dos jogos. Ao fim de cada jogo, isto é, fim de um episódio (linha 14), que é quando o desempenho final do agente é conhecido, sucesso (vitória) ou falha (empate ou derrota), a função *UpdateRating()* da linha 15 atualiza o valor do *rating* para cada um dos casos armazenados no array AC usando uma das 3 estratégias apresentadas na seção 5.5;

4. *Retenção do Caso*: finalmente, este passo é responsável por armazenar na *biblioteca* C todos os casos do array AC (linha 16). O armazenamento é realizado em TT, conforme descrito na seção 5.2.

Algoritmo 3 :Pseudo-código do sistema *ACE-RL-Checkers*

```
 1: procedure ACE_RL_Checkers()
 2: method:
 3:       C ← LoadCaseLibrary();    //biblioteca de casos
 4:       AC ← ∅;                   //casos aplicáveis (cada início de jogo é zerado)
 5: while fim de episódio desconhecido do
 6:    S_t ← BoardCurrent();
 7:    M ← MatchingCases(C ,S_t);
 8:    repeat
 9:       A_{t+1} ← MoveDecision(M, S_t);
10:       ApplyAction(A_{t+1});
11:       S_{t+1} ← NextBoard();
12:    until S_t ≠ S_{t+1}
13:    AC ← AC ∪ Case(S_t, A_{t+1});
14: end while
15: AC ← UpdateRating(AC);
16: Store(C, AC);
```

5.6.1 Mecanismo de Seleção de Ação – *MoveDecision()*

Baseado no conjunto M retornado pelo passo 1 (*recuperação de casos*) da linha 7 do algoritmo 3, a função *MoveDecision()*, ou mecanismo de seleção de ação do *ACE-RL-Checkers*, é responsável por definir a ação A_{t+1} que deverá ser executada sobre o

tabuleiro do jogo corrente S_t. Tal mecanismo de seleção de ação é apresentado através do pseudo-código do algoritmo 4. Como pode ser visto, se o conjunto M é vazio (linha 3 do algoritmo), A_{t+1} será definida pelo *Módulo AR* que é representado pela função ***NN_SuggestMove()*** (linha 4). Tal módulo é responsável por acionar a rede *MLP* do agente estático *LS-VisionDraughts* que, em conjunto com a estratégia de busca *Alfa-Beta* com TT e ID, retornará a melhor ação a ser executada em S_t. Por outro lado, se o conjunto M não é vazio, A_{t+1} será definida pelo *Módulo EAC* que é representado pela função ***ACE_SuggestMove()*** (linha 6). Tal módulo utiliza a versão probabilística da técnica *EAC* proposta por Powell em [21] para retornar a ação mais adequada a ser executada pelo agente em função do conjunto M.

Os módulos *AR* e *EAC* são detalhados nas próximas subseções. Antes, porém, é importante destacar que, na versão original da técnica *EAC* proposta por Powell em [21], sempre que o conjunto M é vazio (linha 3 do algoritmo 4), uma nova ação aleatória é simplesmente executada pelo agente *CHEBR*. Tal estratégia apresenta o incoveniente de produzir uma alta taxa de execução de movimentos aleatórios, fato que compromete o desempenho do agente e a conversão da técnica *EAC*. Com o objetivo de atacar tal problema, o *Módulo AR* é então proposto pelo presente trabalho, conforme indicado na linha 4 do algoritmo 4.

Algoritmo 4 :Pseudo-código da função *MoveDecision*(), mecanismo de seleção de ação do *ACE-RL-Checkers*

```
1: function MoveDecision(var M: matching cases, var S_t: current board): Action
2: method:
3: if M = ∅ then
4:    Return NN_SuggestMove(S_t);
5: else
6:    Return ACE_SuggestMove(M);
7: end if
```

5.6.1.1 Módulo AR

A figura 24 mostra como o processo de escolha de movimento é definido pelo *Módulo AR* do *ACE-RL-Checkers* em função do tabuleiro de jogo corrente S_t. Resumidamente, sempre que a função ***NN_SuggestMove()*** é acionada pelo passo *reuso de casos* do ciclo *SRBC*, fato disparado em função do conjunto M ser vazio, o fluxo dos passos #1 a #5 mostrados na figura 24 é executado dentro da arquitetura do sistema *LS-VisionDraughts*. Primeiramente, o tabuleiro corrente de jogo S_t é apresentado ao *Módulo de Busca* #1 (passo 1 na figura). Esse módulo constrói uma árvore de busca do jogo, cuja raiz é S_t, utilizando o algoritmo *Alfa-Beta* com TT e ID detalhado na seção 4.5. Cada estado correspondente ao nó folha da árvore de busca é convertido na representação baseada em *features* (definida automaticamente pelo melhor indivíduo do *LS-VisionDraughts*), #2, e é apresentado na entrada da *MLP*, #3. A *MLP* avalia cada um dos nós folha da árvore

de busca e retorna um valor de saída (*predição*) que indica o quanto o estado é favorável para o agente. Esse valor é retornado para o algoritmo *Alfa-Beta*, #4, com o objetivo dele apontar qual o melhor movimento a ser executado em S_t, #5. Tal movimento, A_{t+1}, é então retornado pela função ***NN_SuggestMove()*** indicado na linha 4 do algoritmo 4. Para mais detalhes sobre como é realizado a escolha de movimentos pelo agente *LS-VisionDraughts*, veja seção 4.5.

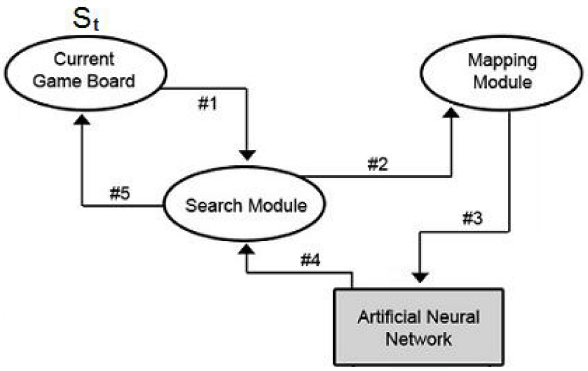

Figura 24 – *Módulo AR* para seleção de uma ação.

5.6.1.2 Módulo *EAC*

O algoritmo 5 mostra o pseudo-código de como o *Módulo EAC* seleciona o caso c_k mais adequado, dentre todos os casos do conjunto M, cuja ação deverá ser executada em S_t. Observe que dentre todos os casos do conjunto M, existe pelo menos um caso que foi gerado por ação sugerida pelo *Módulo AR*, isto é, ação sugerida pelo agente estático *LS-VisionDraughts*. Tal caso foi obtido quando no passado, a *biblioteca* do *ACE-RL-Checkers* não era provida de nenhum caso aplicável ao estado correspondente a S_t.

O dinamismo de escolha de um caso c_k a partir de um conjunto não vazio M é baseado na versão probabilística da técnica *EAC* proposta por Powell em [21]. Em tal dinâmica, a probabilidade de um caso c_k, ou tupla (S_t, A_k), ser selecionado pelo *Módulo EAC* dependerá das experiências de sucesso (vitórias) obtidas pelo agente até o presente momento, ao executar a ação A_k no estado correspondente a S_t. Assim, quando melhor for o resultado da aplicação de uma determinada ação sobre um tabuleiro de jogo particular, maior é a probabilidade do caso associado com essa ação ser selecionado no futuro – devido ao seu alto valor do atributo *rating*. Se, por outro lado, nenhum caso c_k é selecionado pseudo-aleatoriamente, então uma nova ação aleatória é executada. O seguinte processo descreve como a função ***ACE_SuggestMove()*** opera: um caso do conjunto M, conjunto organizado em ordem decrescente dos valores do *rating* dos casos, isto é, $M = \{M_0, M_1, ..., M_n\}$, onde $rating(M_i) \geq rating(M_{i+1})$ e $i \in \{0,1,...,n\text{-}1\}$, é escolhido pseudo-aleatoriamente com o objetivo de encorajar a exploração no espaço de busca. A probabilidade $P(M_0)$ do caso de melhor desempenho (maior valor de *rating*) ser selecionado é igual ao próprio valor do

rating do caso M_0 (linha 5 do algoritmo 5). É importante destacar que para as estratégias *DM* e *GPM*, o *range* de seleção pseudo-aleatória dos casos utilizado foi [0,1], conforme mostrado na própria linha 5 do algoritmo 5. Já para a estratégia *UCM*, o *range* utilizado foi [0,R^{max}], onde R^{max} representa o maior valor de *rating* dos casos armazenados na biblioteca. Se o caso com melhor desempenho M_0 não é selecionado pseudo-aleatoriamente, então ele é removido do conjunto M (linha 8) e a função ***ACE_SuggestMove()*** é chamada novamente (linha 9). Caso contrário, a ação A_{t+1} associada ao caso M_0 é retornada pelo procedimento *ExtractAction()*, linha 6. Se, entretanto, por outro lado, o conjunto inteiro M for percorrido e nenhum caso é selecionado pseudo-aleatoriamente (linha 3), então uma nova ação aleatória é retornada pelo procedimento *NewAction()* da linha 4.

Observe que o *Módulo EAC* de seleção pseudo-aleatória de casos proposto por Powell, em [21], encoraja a exploração de novos casos e o consequente aumento da *base de conhecimentos* do agente.

Algoritmo 5 :Pseudo-código da função *ACE_SuggestMove*() para seleção de uma ação

```
 1: function ACE_SuggestMove(var M: matching cases): Action
 2: method:
 3: if M = ∅ then
 4:    A_{t+1} ← NewAction();
 5: else if Rating(M_0) ≥ Random(0..1) then
 6:    A_{t+1} ← ExtractAction(M_0);
 7: else
 8:    M ← M - M_0;
 9:    A_{t+1} ← ACE_SuggestMove(M);
10: end if
11: Return A_{t+1};
```

5.7 Experimentos e Análise dos Resultados

Esta seção avalia o desempenho do sistema *ACE-RL-Checkers* em relação às duas arquiteturas que motivaram sua construção: arquitetura estática do agente *LS-VisionDraughts* e a arquitetura dinâmica do agente *CHEBR*. Para tanto, esta seção é organizada da seguinte forma. Primeiramente, seção 5.7.1 mostra os resultados obtidos em torneio com a reprodução da versão original do agente *CHEBR*, proposto por Powell em [21], contra o agente *LS-VisionDraughts*. Em seguida, seção 5.7.2 avalia o desempenho em torneio da versão do *ACE-RL-Checkers* que adota a estratégia de atualização do *rating Decaimento de Memória*, estratégia original proposta por Powell em [21] e que foi apresentada na seção 5.5.1. Por fim, seção 5.7.3 avalia o desempenho das versões do *ACE-RL-Checkers* referentes às duas novas estratégias propostas neste trabalho para atualização do *rating* dos casos gerados no contexto da técnica *EAC*, isto é, *Memória Positiva Geral* e *Memória de Confiança Superior*. Tal avaliação considera os mesmos cenários de testes realizados na seção 5.7.2.

5.7.1 Reproduzindo *CHEBR* e avaliando contra *LS-VisionDraughts*

O primeiro experimento realizado pelo presente trabalho foi a reprodução do agente *CHEBR* que utiliza a versão probabilística da técnica *EAC* proposta por Powell em [21]. Tal agente foi submetido a 3 sessões de 1.000 jogos de treinamento contra uma versão do *LS-VisionDraughts* que usa um *look-ahead* com profundidade 4, mesma profundidade adotada por Powell em seus experimentos em [21]. Ao fim de cada sessão de treinamento, 4 jogos testes foram realizados entre os agentes *CHEBR* e *LS-VisionDraughts*. Os resultados do treinamento e torneio são apresentados na tabela 15 em termos de percentuais de vitória, empate e derrota obtidos pelo *CHEBR* em relação ao total de jogos. O resultado mostrado na tabela 15 é o melhor resultado de 3 execuções realizadas, com uma pequena variabilidade nos resultados devido ao uso da abordagem probabilística para seleção pseudo-aleatória de casos por parte do *CHEBR*.

Como pode ser visto na tabela, o agente *CHEBR* não conseguiu superar *LS-VisionDraughts*, tanto nos jogos de treinamento quanto nos jogos de torneio, mesmo o agente *LS-VisionDraughts* tendo sua profundidade de busca limitada a 4 e ter aprendido a jogar Damas com apenas 1.600 jogos de treinamento (para mais detalhes sobre o treinamento desse agente, veja seção 4.4.2). Os resultados, portanto, mostram que a técnica *EAC* proposta em [21] não consegue superar um agente estático não supervisionado treinado por uma eficiente técnica de *AR Diferenças Temporais*. Ao analisar os experimentos, o autor observou que o mecanismo de exploração pseudo-aleatória da técnica *EAC* foi responsável por conduzir *CHEBR*, na maioria das vezes, para vários cenários de jogo (estados de tabuleiro) bastante desfavoráveis para o agente. Esse fato foi agravado pela alta frequência com que ações aleatórias são executadas por *CHEBR*. Com o objetivo de melhor evidenciar tal fato, a tabela 15 mostra o percentual de *novas ações aleatórias* geradas pela técnica *EAC*, em relação ao total de movimentos armazenados na *biblioteca* de casos, ao longo dos 3.000 jogos de treinamento. Observe que o baixo desempenho obtido pelo agente *CHEBR* está diretamente vinculado ao alto índice de *novas ações aleatórias* (93,07%) geradas pela técnica *EAC*.

Nesse sentido, com o objetivo de atacar tal problema apresentado pelo agente *CHEBR*, o sistema híbrido *ACE-RL-Checkers* foi então proposto pelo presente trabalho, conforme arquitetura descrita na seção 5.1. As próximas seções apresentam os experimentos realizados e os ganhos obtidos com a versão híbrida da técnica *EAC* aqui proposta.

5.7.2 Avaliando *ACE-RL-Checkers* com a estratégia original de Powell para atualização do *rating*

O mesmo cenário de testes aplicado ao agente *CHEBR* na seção 5.7.1 também foi aplicado ao sistema *ACE-RL-Checkers* em jogos de treinamento e torneio contra o agente *LS-VisionDraughts*. A tabela 16 mostra os resultados obtidos por *ACE-RL-Checkers* em

Tabela 15 – Treinamento e torneio entre *CHEBR* e *LS-VisionDraughts*.

Jogadores (Match)	Treinamento (3 sessões de 1.000 jogos)			Torneio (4 jogos depois de cada sessão de treinamento)			% Novas Ações Aleatórias (em 3.000 jogos)
	Vitória	Empate	Derrota	Vitória	Empate	Derrota	
CHEBR x LS-VisionDraughts (prof. 4)	0,17%	0,27%	99,56%	8,33%	25,00%	66,67%	93,07%

Resultado do Torneio (CHEBR):

Derrota

termos de percentuais de vitória, empate e derrota em relação ao total de jogos, bem como o percentual de *novas ações aleatórias* geradas pela técnica *EAC*, em relação ao total de movimentos armazenados na *biblioteca* de casos, ao longo dos 3.000 jogos de treinamento. A versão do agente *LS-VisionDraughts* utilizada nos experimentos é a mesma adotada nos resultados apresentados na seção 4.6, isto é, versão sem acesso às bases de final de jogo do *Chinook*, profundidade inicial 4 com aprofundamento iterativo até 8 e 1.600 jogos de treinamento. Como pode ser visto nos resultados da tabela 16, o uso do conhecimento da *MLP* como "identidade base" do sistema *ACE-RL-Checkers*, isto é, o alicerce para a *base de conhecimentos* do *SRBC*, foi determinante para nortear o mecanismo de exploração aleatória da técnica *EAC* de forma a derrotar o jogador *LS-VisionDraughts*. Além disso, tal conhecimento também contribuiu para reduzir significantemente o percentual de *novas ações aleatórias* geradas pelo agente *CHEBR*, conforme resultados apresentados na seção 5.7.1. Veja que com a inclusão do *Módulo AR* apresentado na seção 5.6.1.1, *ACE-RL-Checkers* conseguiu reduzir de 93,07% para apenas 11,49% de *novas ações aleatórias* geradas pela técnica *EAC*.

Tabela 16 – Treinamento e torneio entre *ACE-RL-Checkers* e *LS-VisionDraughts*.

Jogadores (Match)	Treinamento (3 sessões de 1.000 jogos)			Torneio (4 jogos depois de cada sessão de treinamento)			% Novas Ações Aleatórias (em 3.000 jogos)
	Vitória	Empate	Derrota	Vitória	Empate	Derrota	
ACE-RL-Checkers x LS-VisionDraughts (prof. 4 a 8)	36,00%	55,00%	9,00%	41,67%	33,33%	25,00%	11,49%

Resultado do Torneio (ACE-RL-Checkers):

Vitória

O gráfico da figura 25 detalha a evolução do processo de aprendizagem do sistema *ACE-RL-Checkers* ao longo dos 3.000 jogos de treinamento contra *LS-VisionDraughts*. Observe que a arquitetura híbrida precisou um pouco mais de 1.000 jogos de treino para poder superar a arquitetura estática. Vale destacar que a única diferença entre ambas arquiteturas é apenas a inclusão da técnica de aprendizagem *EAC*, que devido a sua natureza pseudo-aleatória possui a habilidade de explorar novas regiões do espaço de busca. Tal habilidade aliada ao conhecimento do próprio agente estático *LS-VisionDraughts* pro-

porcionou que a arquitetura híbrida obtivesse melhor desempenho e maior adaptabilidade em relação a arquitetura estática.

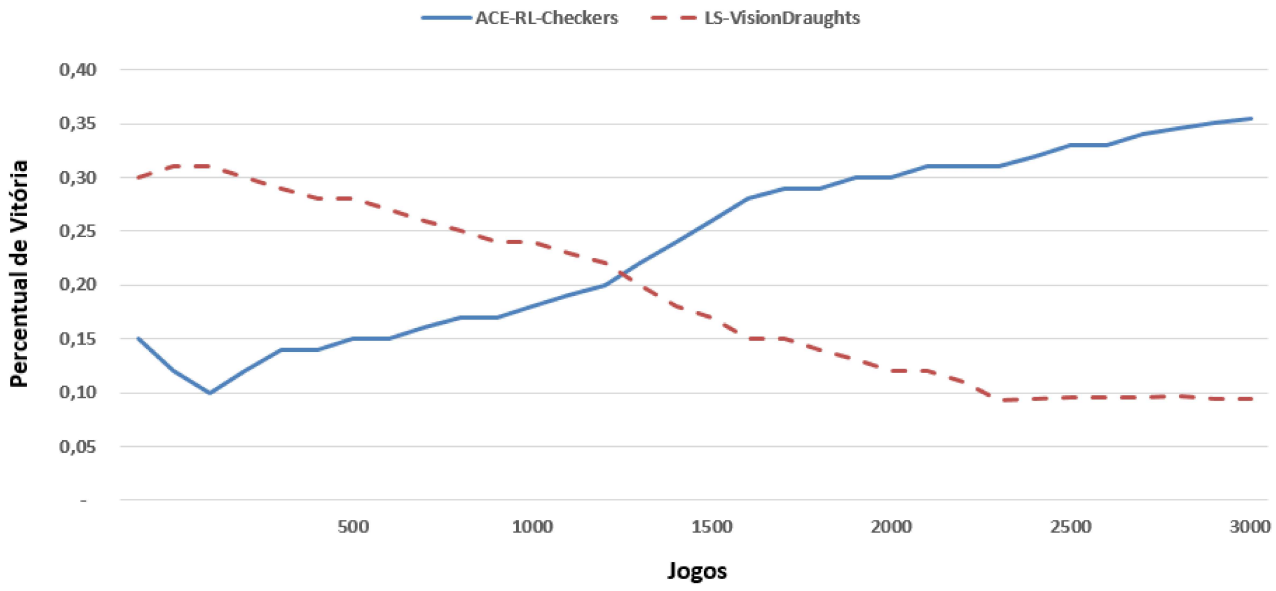

Figura 25 – Percentual de vitória do *ACE-RL-Checkers* em jogos de treinamento contra *LS-VisionDraughts*.

Por fim, com o objetivo de avaliar as duas versões da técnica *EAC* abordadas neste trabalho, isto é, a versão original de Powell [21] e a versão híbrida aqui proposta, a tabela 17 mostra o percentual de vitória do *ACE-RL-Checkers* ao longo de 3.000 jogos de treinamento (incluindo torneio) contra o agente *CHEBR*. Além disso, a tabela 17 também mostra os percentuais de *novas ações aleatórias* geradas pelas duas versões da técnica *EAC*, em relação ao total de movimentos armazenados na *biblioteca* de casos, ao longo dos jogos de treinamento. Como pode ser visto na tabela, a adição do conhecimento da *MLP* foi determinante para que a versão híbrida superasse a versão de Powell, além de ter contribuído para reduzir significantemente o percentual de *novas ações aleatórias* geradas pela técnica *EAC*.

Tabela 17 – Treinamento e torneio entre *ACE-RL-Checkers* e *CHEBR*.

Jogadores (Match)	Treinamento (3 sessões de 1.000 jogos)			Torneio (4 jogos depois de cada sessão de treinamento)			% Novas Ações Aleatórias (em 3.000 jogos)	
	Vitória	Empate	Derrota	Vitória	Empate	Derrota	ACE-RL	CHEBR
ACE-RL-Checkers x CHEBR	98,50%	1,17%	0,33%	100,00%	0,00%	0,00%	8,62%	97,92%

Resultado do Torneio (ACE-RL-Checkers):

Vitória

5.7.3 Avaliando *ACE-RL-Checkers* com as novas estratégias para atualização do *rating*

O primeiro experimento conduzido para investigar as novas estratégias de atualização do *rating* dos casos do *SRBC*, propostas nas seções 5.5.2 e 5.5.3 deste trabalho, foi submeter as 2 versões do sistema híbrido *ACE-RL-Checkers*, cada versão correspondente a uma das duas estratégias investigadas aqui, isto é, *GPM* e *UCM*, em 3 sessões de 1.000 jogos de treinamento contra o agente *LS-VisionDraughts*. Para a versão particular do *ACE-RL-Checkers* que adota a estratégia *UCM* ainda foram criadas mais 5 novas subversões, uma para cada valor constante C utilizado nos experimentos: 0.0, 0.01, 0.05, 0.5 e 1. Tal subdivisão tem como objetivo avaliar o quanto o *trade-off* (ou dilema) entre os fatores *exploration* e *exploitation* influência no desempenho do agente. A versão do agente *LS-VisionDraughts* utilizada nos experimentos é a mesma adotada nos resultados apresentados na seção 4.6, isto é, versão sem acesso às bases de final de jogo do *Chinook*, profundidade inicial 4 com aprofundamento iterativo até 8 e 1.600 jogos de treinamento. Ao fim de cada sessão de treinamento, 4 jogos testes foram realizados entre os sistemas *ACE-RL-Checkers* e *LS-VisionDraughts*. Os resultados do treinamento e torneio são apresentados na tabela 18 em termos de percentuais de vitória, empate e derrota obtidos pelas versões do *ACE-RL-Checkers* em relação ao total de jogos. O resultado mostrado na tabela 18 é o melhor resultado de 3 execuções realizadas, com uma pequena variabilidade nos resultados devido ao uso da abordagem probabilística para seleção pseudo-aleatória de casos por parte do *ACE-RL-Checkers*. É importante destacar que os resultados apresentados na primeira linha da tabela 18, referente à versão *ACE-RL-Checkers* que adota a estratégia *DM*, são os mesmos apresentados na seção 5.7.2 – eles foram mantidos aqui para facilitar o estudo comparativo das estratégias investigadas nesta seção.

Conforme pode ser visto na tabela, o uso do conhecimento da *MLP* como "identidade base" do sistema *ACE-RL-Checkers* para geração da *base de conhecimentos* do *SRBC*, foi determinante para nortear o mecanismo de exploração aleatória da técnica *EAC* de forma que as 3 versões do *ACE-RL-Checkers*, exceto as subversões que adotam a estratégia *UCM* com $C > 0.05$, pudessem derrotar *LS-VisionDraughts*. Vale destacar que a única diferença entre as arquiteturas *ACE-RL-Checkers* e *LS-VisionDraughts* é apenas a inclusão da técnica de aprendizagem *EAC*, que devido a sua natureza pseudo-aleatória possui a habilidade de explorar novas regiões do espaço de busca. Tal habilidade aliada ao conhecimento do próprio agente estático *LS-VisionDraughts* proporcionou que a arquitetura híbrida obtivesse melhor desempenho e maior adaptabilidade em relação a arquitetura estática. Outro ponto que vale ser destacado nos resultados da tabela 18 é o desempenho superior obtido com as versões *ACE-RL-Checkers* que adotam as estratégias *GPM* e *UCM*, esse com $0.0 \geq C \leq 0.01$, em relação ao desempenho da versão que adota a estratégia *DM* – mesma estratégia adotada na versão probabilística da técnica *EAC* proposta por Powell em [21]. Observe que a melhora na acurácia dos valores dos *ratings*

dos casos gerados pelas versões *GPM* e *UCM*, esse com $0.0 \geq C \leq 0.01$, em relação à versão *DM*, pode ser visto através do baixo percentual de *novas ações aleatórias* geradas pela técnica *EAC* e da baixa quantidade total de casos armazenados na *biblioteca SRBC*.

Entretanto, por outro lado, o baixo desempenho obtido com a versão *ACE-RL-Checkers* que adota a estratégia *UCM* com $C > 0.05$ é justificado pelo fato de que à medida que aumenta-se a importância do fator *exploration* da equação *UCM* (maior valor de C), maior prioridade é dada para a exploração de casos pouco utilizados (*exploration*) em comparação com a exploração de casos mais promissores (*exploitation*), isto é, casos com maior média de vitórias. Tal estratégia aliada com a natureza pseudo-aleatória da técnica *EAC* aumenta ainda mais os saltos realizados por tal técnica no espaço de busca. Por isso, quanto mais baixo é a constante C, melhor é o desempenho obtido pelo agente. É importante destacar, porém, que a estratégia do dilema *exploration/exploitation* é útil em domínios complexos como o jogo Go 19×19 por exemplo, onde o espaço de estado é extremamente grande e a obtenção de uma *função de avaliação* bem definida é uma tarefa bastante complicada, daí a necessidade de dar saltos no espaço de busca [69].

Tabela 18 – Treinamento e torneio entre as 3 versões do *ACE-RL-Checkers* e *LS-VisionDraughts*.

ACE-RL-Checkers x LS-Visiondraughts (prof. 4 a 8)	Treinamento (3 sessões de 1.000 jogos)			Torneio (4 jogos depois de cada sessão de treinamento)			% Novas Ações Aleatórias (em 3.000 jogos)	Total de Casos (em 3.000 jogos)
	Vitória	Empate	Derrota	Vitória	Empate	Derrota		
Estratégia DM	36,00%	55,00%	9,00%	41,67%	33,33%	25,00%	11,49%	87.397
Estratégia GPM	99,10%	0,67%	0,23%	100,00%	0,00%	0,00%	7,16%	1.284
Estratégia UCM *C = 0.0*	96,70%	2,10%	1,20%	100,00%	0,00%	0,00%	9,45%	3.240
Estratégia UCM *C = 0.01*	51,80%	35,10%	13,10%	50,00%	33,33%	16,67%	10,27%	63.025
Estratégia UCM *C = 0.05*	30,47%	48,87%	20,66%	33,33%	41,67%	25,00%	29,72%	95.514
Estratégia UCM *C = 0.5*	9,00%	54,67%	36,33%	8,33%	41,67%	50,00%	43,51%	101.968
Estratégia UCM *C = 1.0*	8,94%	53,43%	37,63%	8,33%	50,00%	41,67%	46,20%	101.882

Resultado do Torneio (ACE-RL-Checkers):

Vitória
Derrota

Baseado nos resultados obtidos na tabela 18, o presente trabalho realizou um novo torneio entre as 3 melhores versões do *ACE-RL-Checkers*. Tabela 19 mostra os resultados obtidos pela versão *ACE-RL-Checkers* que adota a estratégia *GPM* em termos de percentuais de vitória, empate e derrota em relação à quantidade total de jogos disputados contra as demais versões *DM* e *UCM* com C = 0.0. Além disso, as medidas de percentual de *novas ações aleatórias* geradas e a quantidade total de casos armazenados na *biblioteca* também são apresentados na tabela – o nome do sistema *ACE-RL-Checkers* foi abreviado

como *ACE-RL*, na tabela 19, com o objetivo de facilitar as notações adotadas pelas 3 versões avaliadas. Conforme pode ser visto, tanto nos jogos de treinamento quanto nos torneios, a versão *ACE-RL-Checkers* que adota a estratégia *GPM* foi superior às demais versões testadas, além de também ter conseguido melhorar a acurácia dos casos gerados pelo agente. Tal fato é demonstrado através do baixo percentual de *novas ações aleatórias* geradas pela técnica *EAC* e da baixa quantidade total de casos armazenados na *biblioteca SRBC*.

Tabela 19 – Treinamento e torneio entre *ACE-RL-Checkers - versão GPM* e *ACE-RL-Checkers - versões DM e UCM*.

Jogadores (Matches)	Treinamento (3 sessões de 1.000 jogos)			Torneio (4 jogos depois de cada sessão de treinamento)			% Novas Ações Aleatórias (em 3.000 jogos)		Total de Casos (em 3.000 jogos)	
	Vitória	Empate	Derrota	Vitória	Empate	Derrota	GPM	DM/UCM	GPM	DM/UCM
ACE-RL GPM X ACE-RL DM	31,93%	48,00%	20,07%	33,33%	50,00%	16,67%	4,04%	10,26%	78.335	112.402
ACE-RL GPM X ACE-RL UCM (C=0.0)	25,77%	56,86%	17,37%	25,00%	66,67%	8,33%	5,50%	8,40%	80.810	110.499

Resultado do Torneio (ACE-RL-Checkers - versão GPM):

Vitória

Por fim, com o objetivo de avaliar as duas versões da técnica *EAC* abordadas neste trabalho, isto é, a versão original de Powell [21] e a versão híbrida aqui proposta, a tabela 20 mostra o percentual de vitória do *ACE-RL-Checkers* ao longo de 3.000 jogos de treinamento (incluindo torneio) contra a versão original *CHEBR*, bem como os percentuais de *novas ações aleatórias* geradas e a quantidade total de casos armazenados na *biblioteca* do *SRBC* para as duas versões da técnica *EAC*. Como pode ser visto na tabela, a adição do conhecimento da *MLP* foi determinante para que todas as estratégias de atualização do *rating* dos casos adotadas pelas versões híbridas pudessem superar a versão de Powell. Além disso, as versões híbridas também reduziram significantemente o percentual de *novas ações aleatórias* geradas pela técnica *EAC* e a quantidade total de casos armazenados na *biblioteca* do *SRBC*.

5.8 Considerações Relativas ao Capítulo

Este capítulo apresentou como a combinação híbrida de uma técnica *EBL* baseada em uma abordagem probabilística em conjunto com um agente estático treinado por *AR* pode conduzir ao desenvolvimento de um sistema dotado da habilidade de, progressivamente, traçar um perfil de seu adversário que o auxilie em seu processo de seleção de ações apropriadas à dinâmica de cada jogo. A técnica *EBL* utilizada foi a versão probabilística da técnica *EAC* proposta por Powell em [21] e o agente estático, utilizado como "identidade base" para o sistema híbrido *ACE-RL-Checkers* aqui proposto, foi o jogador

Tabela 20 – Treinamento e torneio entre as 3 versões do *ACE-RL-Checkers* e *CHEBR*.

ACE-RL-Checkers x CHEBR	Treinamento (3 sessões de 1.000 jogos)			Torneio (4 jogos depois de cada sessão de treinamento)			% Novas Ações Aleatórias (em 3.000 jogos)		Total de Casos (em 3.000 jogos)	
	Vitória	Empate	Derrota	Vitória	Empate	Derrota	ACE-RL	CHEBR	ACE-RL	CHEBR
Estratégia DM	98,50%	1,17%	0,33%	100,00%	0,00%	0,00%	8,62%	97,92%	62.111	91.691
Estratégia GPM	100,00%	0,00%	0,00%	100,00%	0,00%	0,00%	5,79%	99,25%	58.658	90.135
Estratégia UCM *C = 0.0*	100,00%	0,00%	0,00%	100,00%	0,00%	0,00%	7,72%	99,20%	59.829	80.249
Estratégia UCM *C = 0.01*	99,97%	0,03%	0,00%	100,00%	0,00%	0,00%	8,03%	99,24%	60.988	81.385
Estratégia UCM *C = 0.05*	99,90%	0,03%	0,07%	100,00%	0,00%	0,00%	15,79%	99,21%	63.226	101.395
Estratégia UCM *C = 0.5*	99,80%	0,03%	0,17%	100,00%	0,00%	0,00%	21,07%	99,19%	65.644	95.214
Estratégia UCM *C = 1.0*	99,83%	0,03%	0,14%	100,00%	0,00%	0,00%	24,24%	99,22%	65.994	101.524

Resultado do Torneio (ACE-RL-Checkers):

Vitória

LS-VisionDraughts implementado no capítulo 4.

Outro ponto avaliado no presente capítulo foi a investigação de duas novas estratégias alternativas, *Memória Positiva Geral* e *Memória de Confiança Superior*, para calcular o valor do *rating* dos casos gerados no contexto do *ACE-RL-Checkers*.

Os experimentos mostraram que o sistema *ACE-RL-Checkers* é superior às duas versões predecessoras que motivaram sua construção, isto é, os agentes *CHEBR* e *LS-VisionDraughts*. Além disso, a versão *ACE-RL-Checkers* que adota a estratégia *Memória Positiva Geral* mostrou ser superior às demais estratégias de atualização do *rating* avaliadas neste capítulo. Tal estratégia contribuiu para reduzir significantemente o percentual de *novas ações aleatórias* geradas pela técnica *EAC*, bem como a quantidade total de casos armazenados na *biblioteca* do *SRBC*.

Contudo, ao conceber a arquitetura híbrida proposta neste capítulo para o sistema *ACE-RL-Checkers*, é necessário evitar a seguinte fragilidade: nas fases iniciais do jogo em que a quantidade de casos disponíveis na *biblioteca* da técnica *EAC* é extremamente baixa em função do exíguo conhecimento do perfil do adversário, o desempenho do agente é geralmente comprometido pela alta frequência de execução de movimentos aleatórios sobre o tabuleiro. Tal fragilidade ocorre devido a dois motivos: primeiro, porque a *biblioteca de casos* é sempre inicializada (zerada) para cada oponente com o qual o agente interage, isto é, o agente começa jogando sem nenhum conhecimento sobre seu adversário; segundo, devido às características inerentes à técnica *EAC*, a tomada de decisão dinâmica do agente é guiada, ora pelo perfil do adversário – que é quando *casos* são recuperados da *biblioteca*, ora aleatoriamente – que é quando a técnica *EAC* não recupera nenhum caso da *biblioteca*. Essa última situação ocorre em função do próprio mecanismo *EAC* de seleção pseudo-aleatória de casos optar por explorar novas regiões no espaço de busca ou em

função da *biblioteca* não possuir informações suficientes reunidas sobre um determinado perfil de jogo do adversário (situação em que há escassez de casos). Com o objetivo de atacar tal fragilidade, o presente trabalho propõe, na sequência, a inserção de um novo módulo na arquitetura do *ACE-RL-Checkers* composto por uma base de *regras de experiência* minerada, a partir de registros de jogos de especialistas humanos, por meio de uma técnica de *Mineração de Padrões Sequenciais*. Detalhes dessa nova arquitetura são apresentados no capítulo 6.

Capítulo 6

Melhorando o Início de Jogo do ACE-RL-Checkers com *Mineração de Padrões Sequenciais*

Este capítulo apresenta uma versão estendida do sistema *ACE-RL-Checkers* que além de introduzir flexibilidade de tomada de decisão através de um mecanismo que se adapta ao perfil de seu oponente no decorrer de um jogo, lida com a fragilidade do agente associada ao problema do *cold-start* nas fases iniciais do jogo de Damas. Como estratégia de implementação, este trabalho propõe a aplicação de uma técnica de *Mineração de Padrões Sequenciais* para gerar uma base de *regras de experiência*, extraída a partir de registros de jogos de especialistas humanos, com o objetivo de refinar e acelerar o processo de adaptação do agente ao perfil de seu adversário nas fases iniciais de sua interação contra ele. Tal abordagem refere-se ao objetivo 5 traçado na seção 1.3.1.

A principal motivação deste trabalho em aplicar a técnica de *Mineração de Padrões Sequenciais*, proposta por Liang Wang em [2], com o objetivo de melhorar o início de jogo do sistema *ACE-RL-Checkers*, é fundamentada nos resultados obtidos por [2] com a construção de um agente automático de Damas Chinesa baseado apenas em *MPS*. Dentre os resultados obtidos em [2], dois pontos merecem ser destacados: primeiro, o algoritmo *ErsMining* proposto é fácil de ser implementado e demonstrou ser mais eficiente que o tradicional algoritmo *PrefixSpan* – tal algoritmo é bastante utilizado em vários trabalhos que adotam técnicas de *MPS* [85]; segundo, os autores defendem que a nova abordagem de *MPS* é bastante promissora e eficiente, visto que permite aos agentes jogadores uma forma automática de melhorar seu nível de jogo a partir de experiência humana lida de uma base de *log de jogos*. Além disso, os autores também citam que a nova abordagem *MPS* é fácil de ser aplicada para outros tipos de jogos [2].

Por outro lado, uma outra motivação para melhorar o início de jogo do sistema *ACE-RL-Checkers* é baseado no livro *"One Jump Ahead: Computer Perfection at Checkers"* de Jonathan Schaeffer, pesquisador que idealizou a construção do *Chinook*, o qual cita que os

primeiros movimentos no tabuleiro são determinantes para definir o sucesso ou fracasso de um jogador ao longo de uma partida de Damas [77].

Baseado nesses fatos, este capítulo então propõe a construção de um *módulo MPS* para geração de *regras de experiência* de forma que possam ser utilizadas pelo sistema *ACE-RL-Checkers* para melhorar seu nível de jogo inicial, que é quando as tomadas de decisão dinâmica do agente são prejudicadas pela escassez de conhecimento (casos armazenados na *biblioteca*) referente um determinado perfil de jogo. A figura 26 mostra o fluxo de processamento do *módulo MPS* proposto neste trabalho, o qual é composto por três fases principais: *extração de dados*, *mineração de dados* e *representação de dados*. Observe que tal fluxo é bastante similar ao modelo proposto por [2], que foi apresentado na seção 3.3.2.2, mas difere desse nos seguintes aspectos: primeiro, ao invés de utilizar *log de jogos* gerados em tempo real a partir de partidas disputadas por um agente automático contra diversos oponentes, isto é, *log de jogos* gerados *online*, aqui os jogos utilizados referem-se as partidas disputadas por especialistas humanos armazenadas em arquivos no formato *PDN* (do inglês *Portable Draughts Notation*). Tais jogos são, portanto, *offline* e foram coletados a partir de torneios realizados pelas federações *ACF* (*American Checkers Federation*) e *WCDF* (*World Chechers and Draughts Federation*) [108], [109], bem como da base *OCA* (*Open Checkers Archive*) que contem mais de 20.000 jogos compilados por Hans l'Hoest [35]; segundo, o modelo aqui proposto minera apenas *regras de experiência* ou *regras CBSS*. Assim, os demais tipos de *padrões de experiência* apresentados na seção 3.3.2.2 não são considerados neste trabalho; por último, a base de *regras CBSS* mineradas pelo *módulo MPS* é sempre *offline*, ou seja, não sofre qualquer tipo de alteração após o fim de execução de todo o fluxo apresentado na figura 26.

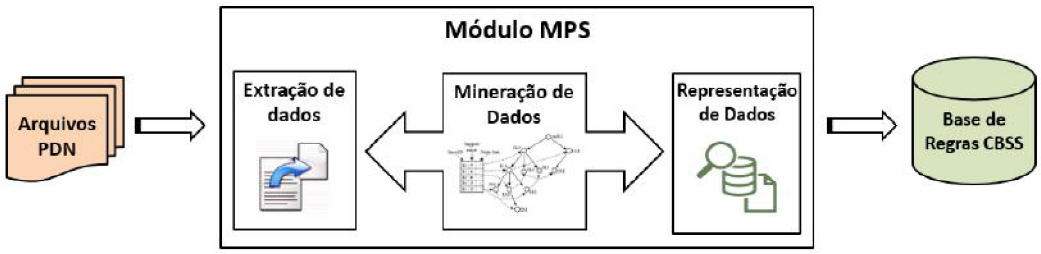

Figura 26 – Fluxo de processamento do *módulo MPS* incorporado ao sistema *ACE-RL-Checkers*.

As próximas seções deste capítulo são organizadas da seguinte forma: seção 6.1 apresenta a arquitetura geral da versão estendida do *ACE-RL-Checkers* com a inclusão do *módulo MPS*; seção 6.2 detalha o *módulo MPS* responsável por minerar a base de *regras de experiência* ou *regras CBSS* a partir de jogos de especialistas; seção 6.3 apresenta o novo mecanismo de seleção de ação do *ACE-RL-Checkers* com o uso da base de *regras CBSS*; por fim, seção 6.4 apresenta os resultados obtidos em torneio com a versão estendida do *ACE-RL-Checkers* proposta neste capítulo.

6.1 Arquitetura Geral da Versão Estendida do *ACE-RL-Checkers*

A figura 27 apresenta a arquitetura geral da versão estendida do sistema *ACE-RL-Checkers* com a inclusão da base de *regras CBSS* mineradas pelo *módulo MPS* (base indicada através do tracejado em verde na figura). Os módulos *EAC* e *AR* da figura compõem o mecanismo de seleção de ação da arquitetura original do *ACE-RL-Checkers* apresentada na seção 5.1 (módulos indicados através do tracejado em azul na figura). Com essa nova arquitetura, o novo processo de tomada de decisão do *ACE-RL-Checkers* pode ser resumido da seguinte forma. Sempre que o agente precisa escolher um novo movimento a partir de um determinado estado de tabuleiro S_t, movimento denominado por a_{t+1}, pois $t+1$ representa um estado futuro ao tempo atual t, o fluxo do processo mostrado através dos passos 1 a 10 da figura 27 é sempre acionado. Primeiramente, o estado de tabuleiro corrente S_t é apresentado ao *Módulo EAC* #1 (passo 1 na figura). Esse módulo utiliza a versão probabilística da técnica *EAC* proposta por [21] para escolher qual o movimento mais adequado a ser executado em S_t. Neste sentido, o *Módulo EAC* tenta recuperar todos os casos (pares estado/ação) que já foram executados no passado pelo agente, no estado S_t, que estão armazenados na *biblioteca* de casos do *SRBC*, #2. Se a tentativa do passo #2 falhar, isto é, o estado S_t é um novo estado para o agente, então o *Módulo EAC* primeiro verifica se o estado S_t refere-se a um estado de tabuleiro inicial de Damas, isto é, qualquer estado que anteceda os 10 primeiros movimentos iniciais executados pelo agente no jogo – a escolha dos 10 primeiros movimentos como referência para determinar início do jogo de Damas foi feita empiricamente, a qual propiciou os melhores resultados sobre uma massa de *jogos testes* em que também foi considerado os 6 e 8 primeiros movimentos iniciais do agente. Se, portanto, S_t referir a um estado de tabuleiro inicial, o *Módulo EAC* tenta recuperar a melhor *regra CBSS* (regra do tipo $S_a \rightarrow S_b$) com maior valor de suporte aplicável para S_t, #3. Se existir tal regra, então os passos #6 a #7 são executados, produzindo o próximo estado de tabuleiro S_{t+1} correspondente ao estado S_b sugerido pela melhor *regra CBSS*. Na sequência, o *Módulo EAC* armazena o novo caso C_{t+1} formado pelo par (S_t,a_{t+1}), que levaram para o estado S_{t+1}, na *biblioteca* de casos do *SRBC*, #8. Se, por outro lado, não existir nenhuma *regra CBSS* aplicável para S_t ou se S_t não referir a um estado de tabuleiro inicial (ainda considerando que o passo #2 falhou), então o *Módulo AR* é acionado, #4. Esse módulo corresponde a *MLP* do melhor indivíduo do *LS-VisionDraughts* e é responsável por escolher a melhor ação a ser executada em S_t utilizando a estratégia de busca em árvore *Alfa-Beta* com TT e ID descrita na seção 4.5. A melhor ação sugerida pelo *Módulo AR*, a_{t+1}, é então retornada para o *Módulo EAC*, #5. Tal ação é executada sobre o tabuleiro S_t, #6, produzindo o estado S_{t+1}, #7, e o caso C_{t+1} formado pelo novo par (S_t,a_{t+1}) é armazenado na *biblioteca* de casos do *SRBC*, #8, sem nenhum valor de *rating* definido. Tal valor só é atualizado após o fim de cada

partida, isto é, após o estado *fim-de-episódio*, que é quando o desempenho final do agente no jogo é conhecido: sucesso (vitória) ou fracasso (empate ou derrota), passos #9 e #10 da figura. É importante destacar que tais passos, #9 e #10, também são aplicados para casos gerados por ações sugeridas pela base de *regras CBSS* durante a fase inicial do jogo. Se, entretanto, a tentativa do passo #2 obter sucesso, isto é, existem casos armazenados aplicáveis a S_t armazenados na *biblioteca* de casos, então os passos #3, #4 e #5 são descartados e o próprio *Módulo EAC* será responsável por escolher pseudo-aleatoriamente a ação a_{t+1} mais adequada a ser executada no passo #6, dentre todos os casos recuperados, produzindo o estado S_{t+1}, #7. O processo de escolha pseudo-aleatória de movimentos do *Módulo EAC* é o mesmo apresentado na seção 5.6.1.2 e pode, inclusive, retornar uma nova ação aleatória, produzindo, assim, um novo caso, daí a necessidade do passo #8 da figura. Os demais passos continuam os mesmos.

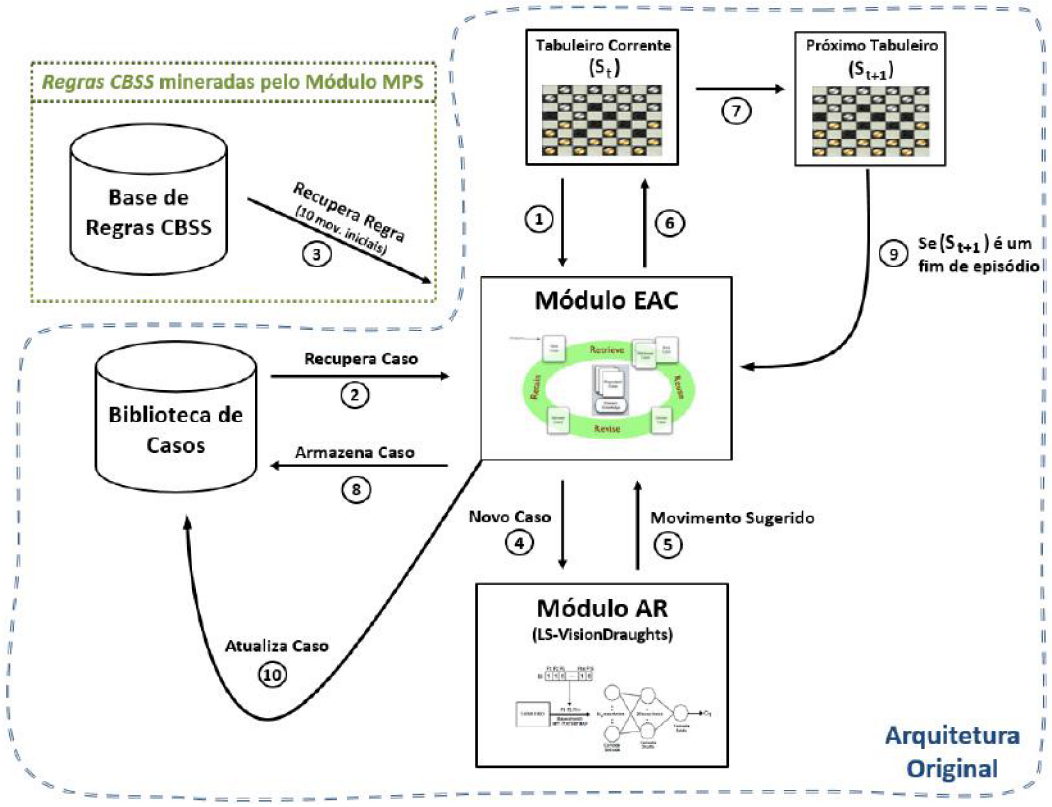

Figura 27 – Arquitetura geral do *ACE-RL-Checkers* com a inclusão da base de *regras CBSS*.

6.2 O Módulo MPS

As próximas seções descrevem cada uma das etapas do fluxo de processamento do *módulo MPS* incorporado à arquitetura do *ACE-RL-Checkers*, conforme desenho apresentado na figura 26.

6.2.1 Processo de Extração de Dados

Nesta etapa foram criadas duas entidades principais de banco de dados, chamadas *BoardState* e *BoardStateSequence*, para armazenar todas as informações necessárias lidas a partir de *log de jogos* armazenados em arquivos PDN. Tais informações são utilizadas pela próxima etapa do *Módulo MPS*, isto é, *processo de mineração de dados*, para gerar as *regras CBSS* que são incorporadas ao sistema *ACE-RL-Checkers*. A figura 28 mostra um exemplo de arquivo PDN que contém o registro de uma partida de Damas disputada entre os jogadores Tinsley e Chamblee no ano de 1946. Como pode ser visto na figura, o arquivo contém informações técnicas em relação ao torneio, tais como, data e nome do evento, quem joga com as peças pretas ou brancas do tabuleiro e o resultado da partida, bem como a sequência de movimentos realizados por ambos jogadores ao longo da partida inteira. Portanto, partindo-se de um tabuleiro inicial de Damas 8×8 e reproduzindo a sequência de movimentos executados por ambos jogadores é possível extrair todas as informações necessárias para carregar as duas entidades propostas nesta fase, isto é, *BoardState* e *BoardStateSequence*.

```
[Event "US National 1946"]
[Black "MF Tinsley"]
[White "M Chamblee"]
[Result "0-1"]
[Date "1946-02-10"]
1. 10-14 22-17 2. 14-18 23x14 3. 9x18 26-23 4. 6-9 23x14 5. 9x18 30-26 6. 2-6
26-23 7. 6-9 23x14 8. 9x18 31-26 9. 5-9 17-13 10. 11-15 13x6 11. 1x10 25-22
12. 18x25 29x22 13. 10-14 27-23 14. 8-11 24-19 15. 15x24 28x19 0-1
```

Figura 28 – Exemplo de arquivo PDN com registro de uma partida disputada entre os jogadores Tinsley e Chamblee.

Os principais atributos das entidades *BoardState* e *BoardStateSequence* são apresentados, respectivamente, nas tabelas 21 e 22. Um exemplo de valores para cada entidade pode ser visto a seguir:

- *BoardState*: “250 | 01000001400012000010220002002030”, onde “250” representa o identificador único do estado de tabuleiro na base de dados de sequência do *Módulo MPS* e “01000001400012000010220002002030” representa o próprio estado de tabuleiro através das 32 posições do tabuleiro de Damas com as seguintes opções de valores: “0” representa uma posição vazia, “1” representa uma peça simples preta, “2” representa uma peça simples branca, “3” representa uma dama preta e “4” representa uma dama branca;
- *BoardStateSequence*: “4 | WW | 0#240#241#...#250#251#252#...#270#271”, onde “4” representa o identificador único da sequência na base de dados do *Módulo MPS*, “WW” representa o resultado do jogo (nesse caso o jogador branco vence) e “0#240#241#...#250#251#252#...#270#271” representa a sequência de todos os estados de tabuleiro de uma partida completa de Damas separados pelo simbolo “#”;

Tabela 21 – Atributos da entidade *BoardState*.

Nome do Atributo	Descrição	Tipo de Dado	Restrições
ID	O identificador único de cada estado de tabuleiro	Inteiro	Único, Não Nulo
BoardStr	*Array* contendo todas as 32 posições do tabuleiro de Damas	Texto	Não Nulo

Tabela 22 – Atributos da entidade *BoardStateSequence*.

Nome do Atributo	Descrição	Tipo de Dado	Restrições
ID	O identificador único da sequência do jogo	Inteiro	Único, Não Nulo
GameResult	O resultado do jogo :: D - empate, BW - preto ganha, WW - branco ganha	Texto	Não Nulo
SequenceStr	*Array* da sequência do jogo	Texto	Não Nulo

Para esta etapa, 4.023 jogos de especialistas humanos foram lidos de arquivos PDN, incluindo jogos de um dos maiores jogadores de Damas da história, Marion Tinsley [15]. Tais jogos foram distribuídos da seguinte forma: 1.112 jogos em que o jogador preto ganha, 1.368 jogos em que o jogador branco ganha e 1.543 empates. Desses jogos, foram extraídos um total de 160.272 tabuleiros distintos para a entidade *BoardState* e 4.023 sequências para a entidade *BoardStateSequence*.

6.2.2 Processo de Mineração de Dados

O objetivo desta etapa é minerar *regras CBSS* a partir das informações armazenadas nas entidades *BoardState* e *BoardStateSequence* carregadas na etapa anterior, i.e, *processo de extração de dados*. Considerando, além disso, que cada *regra CBSS* é composta por dois estados de tabuleiro que são consecutivos na entidade *BoardStateSequence*, o algoritmo *MPS* implementado nesta etapa usa *árvore de sequência* (do inglês *sequence-tree*) como estrutura de dados para carregar todas as *subsequências binárias consecutivas* a partir da base de dados sequenciais carregada na seção 6.2.1. A figura 29 mostra a estrutura de dados utilizada para representar um nó da *árvore de sequência*, o qual contém as seguintes informações:

- *BOARD ID*: identificador único de um estado de tabuleiro;
- *suppCount*: contagem de frequência para cálculo do suporte relativo à *regra CBSS*;
- *whoPlayed*: este atributo indica qual jogador executou o movimento relativo à *regra CBSS*, isto é, o jogador preto ou branco. Essa marcação é importante porque no jogo de Damas existe o problema de transposição de tabuleiro, onde dentro de uma mesma partida pode-se chegar a um mesmo tabuleiro (estado) várias vezes e de diferentes formas [76];
- *next*: um ponteiro para o próximo nó. Note que é esse ponteiro que define a ligação entre dois estados de tabuleiro, que são consecutivos na entidade *BoardStateSequence*, referentes a uma determinada *regra CBSS* do tipo $S_a \rightarrow S_b$, onde S_a representa o estado atual (origem) e S_b representa o próximo estado (destino).

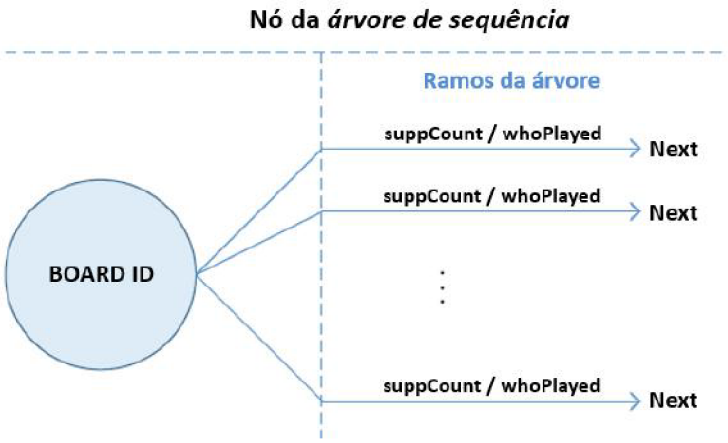

Figura 29 – Estrutura de dados de um nó da *árvore de sequência*.

O algoritmo 6 mostra o pseudo-código da função *SequenceTreeCreation* que é responsável por criar a *árvore de sequência* a partir das entidades *BoardState* e *BoardStateSequence* carregadas na seção 6.2.1. Tal algoritmo faz parte da *engine* do algoritmo *ErsMining* proposto por Liang Want em [2]. *SequenceTreeCreation* recebe como parâmetro de entrada a entidade *BoardStateSequence* e retorna como saída uma *árvore de sequência* com os valores de suporte atualizados para cada *regra CBSS*. Resumidamente, para cada sequência de tabuleiros lida da entidade *BoardStateSequence*, as linhas 4 a 50 do algoritmo são processadas com o objetivo de construir a *árvore de sequência*. A linha 5 extrai os *IDs* dos tabuleiros de uma sequência inteira e joga no array *stateIDs*. As linhas 8 a 15 do algoritmo são responsáveis por remover redundâncias de estados de tabuleiro que aparecem mais de 1 vez em uma única sequência do jogo. Em outras palavras, sob a perspectiva de um determinado jogador, se dois mesmos estados de tabuleiro aparecem em uma sequência, os movimentos entre esses dois estados tendem a ser considerados pouco efetivos para o jogador se comparado com os demais movimentos realizados no jogo. Eliminando, portanto, esses dados, o algoritmo ganhará em tempo de processamento, pois reduzirá as análises de dados desnecessários da base e, consequentemente, melhorará a confiabilidade dos resultados de mineração obtidos [2]. As linhas 19 a 23 são responsáveis por definir um valor de contagem de frequência positivo para *subsequências binárias consecutivas* geradas a partir de jogos que resultaram em vitória ou empate, e negativo caso contrário. O objetivo em adotar tal estratégia é reforçar (positivamente) ou penalizar (negativamente) *regras CBSS* com base em suas frequências de execução aliados aos resultados obtidos nos jogos lidos. As linhas 24 a 47 são responsáveis por construir os nós e ramos da *árvore de sequência* e atribuir as informações referentes a cada atributo do nó, conforme estrutura de dados apresentada na figura 29. É importante destacar aqui que o atributo *whoPlayed* só é preenchido na criação de um novo nó, quando é necessário conhecer qual jogador conduziu o movimento que levou para o estado de tabuleiro resultante (tabuleiro destino) associado à *regra CBSS* (linha 42 do algoritmo). Por fim, a linha 49 do algoritmo

6 realiza um procedimento descrito por [2] como *unificação de nós* ou *branch shifting*. O objetivo de executar tal procedimento é evitar que mesmos pares de nós da árvore, ou seja, mesmas *regras CBSS*, possam aparecer em várias posições da *árvore de sequência*, fato que tornaria o cálculo do suporte relativo das *regras CBSS* (cálculo a ser apresentado na sequência) bastante oneroso, visto que a árvore deveria ser percorrida mais de uma vez. Um exemplo de *unificação de nós* é mostrado na figura 30, onde dois pares de nós $B \rightarrow C$, problema indicado através do tracejado em vermelho na árvore da esquerda, precisam ser unificados e os valores do parâmetro *suppCount* sumarizados. A árvore à direita da figura 30 é o resultado da aplicação do procedimento *BranchShifting*, conforme pseudo-código apresentado no algoritmo 7.

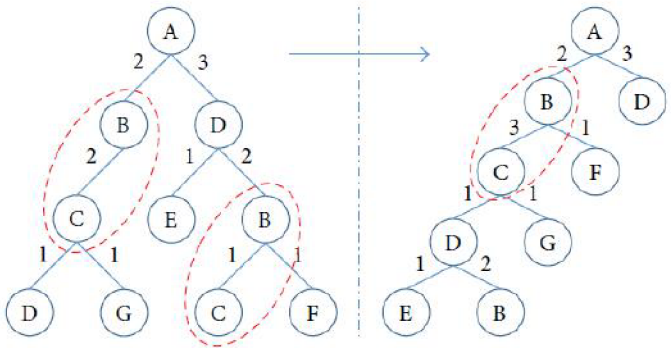

Figura 30 – Um exemplo de *unificação de nós* aplicado a uma *árvore de sequência*.

Para obter as frequências de todas as *regras CBSS*, a seguinte equação é adotada:

$$supp = \frac{suppCount}{totalSeqs} \times 100, \tag{10}$$

onde *supp* representa o suporte relativo de uma *subsequência binária consecutiva* e *totalSeqs* representa a quantidade total de sequências do banco de dados. Assim, dado um determinado *minSupport*, as *regras CBSS* frequentes são aquelas cujos suportes relativos (*supp*) não são menores do que o *minSupport*. Dessa forma, todas as *regras CBSS* frequentes podem ser encontradas simplesmente percorrendo a *árvore de sequência* apenas uma vez, conforme pseudo-código apresentado nas linhas 3 a 17 do do algoritmo 8. Entretanto, para realizar tal procedimento, uma nova estrutura de dados, chamada *SequentialPattern*, foi criada para armazenar todas as *regras CBSS* frequentes mineradas. A figura 31 mostra a estrutura de dados *SequentialPattern*, a qual contém as seguintes informações:

- *Premise ID*: identificador único do estado de tabuleiro origem;
- *Result ID*: identificador único do estado de tabuleiro destino;
- *supp*: atributo que armazena o suporte relativo da *regra CBSS* frequente;
- *whoPlayed*: atributo que indica qual jogador executou o movimento relativo à *regra CBSS*.

Algoritmo 6 :Pseudo-código do algoritmo *SequenceTreeCreation*() responsável por criar uma estrutura de *árvore de sequência* a partir das entidades de banco *BoardState* e *BoardStateSequence*

```
1: function SequenceTreeCreation(sequencesD: a tabela da entidade BoardStateSequence): root
   //nó raiz da árvore de sequência
2: method:
3: root = new Node(boardID = 0, branches = null);
4: for i = 0 to sequencesD.length − 1 do
5:    stateIds = sequencesD[i].SequenceStr.split("#"); //armazena os ids da entidade BoardState de uma
      sequência inteira;
6:    extract qual jogador venceu o jogo e salve na variável whoWon;
7:    //Elimina redundância de estados de tabuleiro da variável stateIds.
8:    for j = stateIds.length − 1 to 2 do
9:       for k = j − 2 to 0 by 2 do
10:         if stateIds[j]==stateIds[k] then
11:            stateIds.delete(k+1,j);
12:            j = k;
13:         end if
14:      end for
15:   end for
16:   node = root;
17:   for m = 1 to stateIds.length − 1 do
18:      extract qual jogador executou o próximo movimento e salve na variável whoPlay;
19:      if whoPlay == whoWon or whoWon corresponde a um empate then
20:         suppAux = 1;
21:      else
22:         suppAux = -1;
23:      end if
24:      if node.boardID == 0 then
25:         node.boardID = stateIds[0];
26:      end if
27:      isFound = false;
28:      if node.branches != null then
29:         for n = 0 to node.branches.length − 1 do
30:            if node.branches[n].next.boardID == stateIds[m] then
31:               node.branches[n].suppCount += suppAux;
32:               node = node.branches[n].next;
33:               isFound = true;
34:               break;
35:            end if
36:         end for
37:      end if
38:      if isFound == false then
39:         branch = new Branch (
40:               suppCount = suppAux,
41:               //Atualiza o jogador que conduziu para esse estado.
42:               whoplay = whoPlay,
43:               next = new Node (boardID = stateIds[m], branches = null);
44:         );
45:         node.branches.add(branch);
46:         node = node.branches[node.branches.length-1].next;
47:      end if
48:      //Unifica nós da árvore referentes às mesmas regras CBSS.
49:      BranchShifting(root, node);
50:   end for
51: end for
52: return root
```

Algoritmo 7 :Pseudo-código do algoritmo *BranchShifting*() responsável por unificar nós da árvore que contém mesmas *regras CBSS*

```
1: function BranchShifting(root: nó raiz da árvore de sequência, node: ponteiro
   para o nó corrente da árvore): isShifted //se houve ou não unificação de nós
2: method:
3: if node.value == root.value and root.branches.length > 0 then
4:    node = root;
5:    return true;
6: else
7:    if root.branches == null then
8:       return false;
9:    else
10:      for i = 0 to root.branches.length − 1 do
11:         if BranchShifting(root.branches[i].next, node) == true then
12:            return true;
13:         end if
14:      end for
15:      return false;
16:   end if
17: end if
```

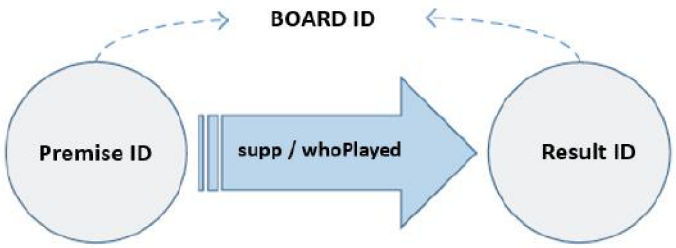

Figura 31 – Estrutura de dados *SequentialPattern* que armazena as *regras CBSS* frequentes mineradas.

É, portanto, em cima da estrutura *SequentialPattern* que a próxima fase do *Módulo MPS*, isto é, *processo de representação de dados*, gerará a base de *regras CBSS* frequentes que será utilizada pelo sistema *ACE-RL-Checkers* para melhorar sua habilidade no início do jogo de Damas.

Nesta etapa foi adotado um *minSupport* de 2%, minerando ao todo 10.074 *regras CBSS* a partir das 4.023 sequências de jogos lidas de arquivos PDN. O tempo total gasto para gerar tais regras mineradas foi de aproximadamente 3 horas em um processador Intel Quad Core i7-2670QM (6M Cache e 2.20 GHz) com 8 GB de memória RAM DDR2. É importante destacar que as 10.074 *regras CBSS* mineradas referem-se as jogadas subsequentes consecutivas realizadas por especialistas humanos ao longo de todo o jogo de Damas e não apenas para as jogadas iniciais sobre o tabuleiro. O autor preferiu adotar tal estratégia com o propósito de investigar a base completa de *regras CBSS* mineradas em trabalhos futuros. Nesse sentido, a restriçao para a utilização das *regras CBSS* apenas

Algoritmo 8 :Pseudo-código do algoritmo *CBSSFinding*() responsável por retornar todas as *regras CBSS* frequentes a partir de uma *árvore de sequência*

```
1: procedure CBSSFinding(minSupport: suporte mínimo informado pelo cliente,
   root: nó raiz da árvore de sequência, total: quantidade total de sequências da base,
   sps: estrutura de dados contendo as regras CBSS frequentes)
2: method:
3: if root.branches == null then
4:    return;
5: end if
6: for i = 0 to root.branches.length − 1 do
7:    supp = root.branches[i].suppCount/total;
8:    if supp ≥ minSupport then
9:       sp = new SequentialPattern();
10:      sp.premise = root.boardID;
11:      sp.result = root.branches[i].next.boardID;
12:      sp.support = supp;
13:      sp.whoPlayed = root.branches[i].whoPlayed;
14:      sps.add(sp);
15:   end if
16:   CBSSFinding(minSupport, root.branches[i].next, total, sps);
17: end for
```

no início do jogo de Damas foi aplicada somente no novo módulo de tomada de decisão do sistema *ACE-RL-Checkers*, conforme será explicado na seção 6.3.

6.2.3 Processo de Representação de Dados

Nesta fase, a tarefa principal do *Módulo MPS* é tornar as *regras CBSS*, mineradas na etapa anterior, "reconhecíveis" para o agente automático implementado neste trabalho. Para tanto, os dois passos a seguir foram realizados:

1. Todos os identificadores de tabuleiro (*IDs*) da estrutura *SequentialPattern*, apresentada na figura 31, foram substituídos pela estrutura *BOARD*, apresentada na seção 5.2, correspondente à representação matricial dos estados de tabuleiro do jogo, seguindo o padrão "$PremiseBOARD \rightarrow ResultBOARD : supp|whoPlayed$";

2. Na sequência, todas as *regras* foram armazenadas, em arquivos específicos, agrupadas por $PremiseBOARD$ e ordenadas pelo valor descrescente de suporte.

Assim, para retornar a melhor *regra CBSS* aplicável a um determinado tabuleiro de jogo S_i, basta recuperar da base de arquivos contendo *regras CBSS* a primeira *regra* que satisfaz as duas restrições a seguir: "$PremiseBOARD == S_i$" e tal *regra* ter sido executada com as mesmas peças do tabuleiro de Damas (fato indicado através do atributo *whoPlayed*) que o agente está jogando no exato momento da consulta, isto é, como jogador preto ou branco. São, portanto, esses arquivos, gerados nesta etapa de *representação de*

dados, que foram incorporados à arquitetura do *ACE-RL-Checkers* para serem consultados pelo sistema nas fases iniciais do jogo de Damas.

6.3 O Novo Mecanismo de Seleção de Ação

O novo mecanismo de seleção de ação da versão estendida do sistema *ACE-RL-Checkers*, proposta neste capítulo, é apresentado através das linhas 3 a 9 do pseudo-código do algoritmo 9 e pode ser resumido a seguir. Baseado no conjunto M retornado pelo passo *recuperação de casos* do ciclo *SRBC* apresentado na seção 5.6, a função *MoveDecision()*, ou mecanismo de seleção de ação do *ACE-RL-Checkers*, é responsável por definir a ação A_{t+1} que deverá ser executada sobre um determinado tabuleiro corrente do jogo S_t. Primeiramente, o sistema testa se as três condições a seguir são satisfeitas (linha 3 do algoritmo 9): 1) o conjunto M é vazio; 2) o estado S_t corresponde a um estado de tabuleiro inicial de Damas, isto é, qualquer estado que antecede os 10 primeiros movimentos iniciais executados pelo agente no jogo; 3) existem *regras* aplicáveis para o estado S_t armazenadas na base de arquivos contendo *regras CBSS*. Se sim, então a ação A_{t+1} será definida pela função ***CBSS_SuggestMove()*** (linha 4) que simplesmente retorna a ação associada à melhor *regra CBSS* do tipo $S_a \rightarrow S_b$ aplicável para S_t seguindo as restrições explicadas na seção 6.2.3. Se, entretanto, não existir nenhuma *regra CBSS* aplicável para S_t ou S_t não refere a um estado de tabuleiro inicial, ainda considerando que o conjunto M é vazio (linha 5), A_{t+1} será definida pelo *Módulo AR* que é representado pela função ***NN_SuggestMove()*** (linha 6). Tal módulo é responsável por acionar a rede *MLP* do agente estático *LS-VisionDraughts* que, em conjunto com a estratégia de busca *Alfa-Beta* com TT e ID, retornará a melhor ação a ser executada em S_t, conforme processo descrito na seção 5.6.1.1. Por outro lado, se o conjunto M não é vazio, A_{t+1} será definida pelo *Módulo EAC* que é representado pela função ***ACE_SuggestMove()*** (linha 8). Tal módulo utiliza a versão probabilística da técnica *EAC* proposta por Powell em [21] para retornar a ação mais adequada a ser executada pelo agente em função do conjunto M, conforme processo descrito na seção 5.6.1.2.

Observe que o novo mecanismo de seleção de ação proposto neste trabalho para o sistema *ACE-RL-Checkers* combina o conhecimento de especialistas, representado na forma de *regras CBSS*, com o conhecimento de um agente estático, representado pela *MLP* do *LS-VisionDraughts*, para direcionar o processo de exploração pseudo-aleatória da técnica *EAC* para regiões mais promissoras no espaço de busca, de forma a acelerar e refinar o processo de adaptação do agente ao perfil de seu adversário nas fases iniciais do jogo de Damas.

Algoritmo 9 :Pseudo-código da nova função *MoveDecision*(), mecanismo de seleção de ação da versão estendida do *ACE-RL-Checkers*

```
1: function MoveDecision(var M: matching cases, var S_t: current board): Action
2: method:
3: if M = ∅ and IsInitialBoard(S_t) and IsThereCBSSRule(S_t) then
4:    Return CBSS_SuggestMove(S_t);
5: else if M = ∅ then
6:    Return NN_SuggestMove(S_t);
7: else
8:    Return ACE_SuggestMove(M);
9: end if
```

6.3.1 Equação de Atualização do *Rating*

A equação de atualização do valor do *rating* dos casos, gerados no contexto da nova versão híbrida da técnica *EAC*, utilizada neste capítulo é a estratégia *Memória Positiva Geral* apresentada na seção 5.5.2. Tal estratégia apresentou os melhores resultados, conforme análises realizadas na seção 5.7.

6.4 Experimentos e Análise dos Resultados

Esta seção avalia o desempenho da versão estendida do sistema *ACE-RL-Checkers* proposta neste capítulo em relação a quatro tipos de análises. Tais análises foram realizadas a partir de torneios envolvendo o agente *LS-VisionDraughts* e as duas versões do *ACE-RL-Checkers* investigadas neste trabalho: a versão estendida que utiliza *regras CBSS*, versão que será referenciada daqui em diante como *ACE-RL com CBSS*, e a melhor versão apresentada no capítulo 5 que adota a estratégia de atualização do valor do *rating* *Memória Positiva Geral* – tal versão também será, analogamente referenciada, daqui em diante, como *ACE-RL sem CBSS*. A seção 6.4.1, primeiramente, investiga o desempenho das duas versões do *ACE-RL-Checkers* contra o agente *LS-VisionDraughts* em termos de acurácia dos casos gerados pela técnica *EAC* e o tempo de treinamento. Seção 6.4.2 apresenta o desempenho obtido em torneio das versões *ACE-RL com CBSS* e *ACE-RL sem CBSS* em termos de percentuais de vitória. Na sequência, seção 6.4.3 tem como objetivo estimar a taxa média de coincidência entre os primeiros movimentos escolhidos pela versão *ACE-RL com CBSS* e seus oponentes, em jogos reais entre eles e os movimentos que, nas mesmas situações, seriam indicados pelo "*conselheiro de movimento*" do forte agente supervisionado *Cake*. Por fim, seção 6.4.4 usa o método de estatística não-paramétrica *Wilcoxon* para validar estatisticamente o ganho real obtido pela versão *ACE-RL com CBSS* nos testes realizados nas seções 6.4.2 e 6.4.3.

6.4.1 Avaliando a Acurácia dos Casos e Tempo de Treinamento

O primeiro experimento realizado pelo autor foi submeter a versão *ACE-RL com CBSS* descrita na seção 6.1 para o mesmo cenário de teste aplicado para a versão *ACE-RL sem CBSS* na seção 5.7.3. O objetivo com tal experimento é avaliar o quanto a inclusão da base de *regras CBSS*, na fase inicial de jogo do agente, tem de fato contribuído para melhorar a acurácia dos casos gerados pela técnica *EAC* e consequentemente, seu processo de treinamento geral. Basicamente, o cenário de teste consiste em submeter as versões *ACE-RL com CBSS* e *ACE-RL sem CBSS* para 3 sessões de 1.000 jogos de treinamento contra o agente *LS-VisionDraughts*. A versão do agente *LS-VisionDraughts* utilizada nos experimentos é a mesma adotada nos resultados apresentados na seção 4.6, isto é, versão sem acesso às bases de final de jogo do *Chinook*, profundidade inicial 4 com aprofundamento iterativo até 8 e 1.600 jogos de treinamento. Ao fim de cada sessão de treinamento, 4 jogos testes foram realizados entre as duas versões do *ACE-RL-Checkers* e *LS-VisionDraughts*. Os resultados do treinamento e torneio são apresentados na tabela 23-a) em termos de percentuais de vitória, empate e derrota obtidos pelas versões do *ACE-RL-Checkers* em relação ao total de jogos. Tabela 23-b) mostra os dados estatísticos obtidos pelas duas versões híbridas da técnica *EAC* em relação ao tempo total de treinamento, quantidade total de ações sugeridas pela base de *regras CBSS* e pela *MLP*, quantidade total de ações aleatórias geradas pela técnica *EAC* e o total de casos armazenados em memória. O resultado mostrado na tabela 23 é o melhor resultado de 3 execuções realizadas, com uma pequena variabilidade nos resultados devido ao uso da abordagem probabilística para seleção pseudo-aleatória de casos por parte do *ACE-RL-Checkers*. É importante destacar que os resultados apresentados na primeira linha da tabela 23-a), referente à versão *ACE-RL-Checkers* que adota a estratégia *GPM*, são os mesmos apresentados na seção 5.7.3 – eles foram mantidos aqui para facilitar o estudo comparativo das abordagens investigadas nesta seção.

Como pode ser visto na tabela 23-a), apesar das duas versões do *ACE-RL-Checkers* serem bastante superiores ao agente *LS-VisionDraughts*, é possível verificar uma pequena melhora na fase de treinamento da versão *ACE-RL com CBSS*. Veja que com essa versão, o agente não perde nenhuma partida e consegue melhorar o percentual de vitórias em relação ao total de jogos. A tabela 23-b) mostra com mais detalhes os dados estatísticos obtidos pelas duas abordagens da técnica *EAC* ao longo dos 3.000 jogos de treinamento. Veja que a versão proposta neste capítulo reduz em 73,68% o tempo total de treinamento gasto em relação a versão *ACE-RL sem CBSS*. Além disso, o uso da base de *regras CBSS* contribuiu para direcionar melhor a exploração pseudo-aleatória da técnica *EAC* para regiões mais promissoras no espaço de busca, gerando, consequentemente, casos mais precisos. Tal comportamento pode ser visto na tabela 23-b) através dos seguintes indicadores: com apenas 7 *regras CBSS* utilizadas no início do jogo, o sistema superou o desempenho obtido pela versão *ACE-RL sem CBSS*, gerando uma quantidade bem mais

Tabela 23 – a) Treinamento e torneio entre *ACE-RL-Checkers* e *LS-VisionDraughts*; b) Resultado do treinamento em termos de acurácia dos casos e tempo de treinamento.

Jogadores (matches)	Treinamento (3 sessões de 1.000 jogos)			Torneio (4 jogos depois de cada sessão de treinamento)		
	Vitória	Empate	Derrota	Vitória	Empate	Derrota
ACE-RL sem CBSS x LS-VisionDraughts (4 to 8-ply)	99,10%	0,67%	0,23%	100%	0,00%	0,00%
ACE-RL com CBSS x LS-VisionDraughts (4 to 8-ply)	99,83%	0,17%	0,00%	100%	0,00%	0,00%

Resultado do Torneio (ACE-RL): Vitória

(a)

Jogadores (matches)	Tempo Total (min) (em 3.000 jogos)	Total de Ações Sugeridas pelas *regras CBSS* (em 3.000 jogos)	Total de Ações Sugeridas pela MLP (em 3.000 jogos)	Total de Ações Aleatórias (em 3.000 jogos)	Total de Casos (em 3.000 jogos)
ACE-RL sem CBSS x LS-VisionDraughts (4 to 8-ply)	38	0	1.192	92	1.284
ACE-RL com CBSS x LS-VisionDraughts (4 to 8-ply)	10	7	560	14	581

(b)

baixa de ações aleatórias e casos armazenados em memória, além de também reduzir a necessidade do uso do conhecimento proveniente da rede *MLP*.

6.4.2 Desempenho nos Torneios

Com o objetivo de avaliar as duas versões do sistema *ACE-RL-Checkers* abordadas neste trabaho, isto é, *ACE-RL com CBSS* e *ACE-RL sem CBSS*, tabela 24-a) mostra o percentual de vitória da versão que adota a base de *regras CBSS*, ao longo de 3.000 jogos de treinamento (incluindo torneio) contra a versão *ACE-RL sem CBSS*. Como pode ser visto na tabela, a inclusão da base de *regras CBSS* foi crucial para melhorar o início de jogo do sistema *ACE-RL-Checkers* de forma a superar sua versão predecessora, isto é, sem acesso a base de *regras CBSS*. Note que a única diferença entre ambas versões do sistema *ACE-RL-Checkers* é apenas a inclusão da base de *regras CBSS* na fase inicial do jogo. Além disso, a tabela 24-b) mostra os dados estatísticos obtidos pelas duas versões da técnica *EAC* durante os 3.000 jogos de treinamento. Veja que com apenas 337 *regras CBSS* utilizadas no início do jogo, a versão estendida superou a versão predecessora utilizando uma quantidade bem mais baixa de ações aleatórias, total de casos armazenados em

memória e total de ações sugeridas pela *MLP*. Esses números comprovam que o uso de uma base de *regras CBSS* no início do jogo de Damas ajuda o agente a melhorar seu desempenho em relação a dois aspectos: melhor direcionamento na exploração pseudo-aleatória da técnica *EAC* para regiões mais promissoras no espaço de busca e geração de casos mais precisos no processo de adaptação do agente ao perfil de seu adversário – fato que pode ser verificado através do melhor desempenho obtido nos torneios por parte da versão *ACE-RL com CBSS*, conforme resultados apresentados na tabela 24-a).

Tabela 24 – a) Treinamento e torneio entre as versões *ACE-RL com CBSS* e *ACE-RL sem CBSS*; b) Resultado do treinamento em termos de acurácia dos casos.

Jogadores (match)	Treinamento (3 sessões de 1.000 jogos)			Torneio (4 jogos depois de cada sessão de treinamento)		
	Vitória	Empate	Derrota	Vitória	Empate	Derrota
ACE-RL com CBSS x ACE-RL sem CBSS	58,17%	26,53%	15,30%	66,67%	25,00%	8,33%

Resultado do Torneio (ACE-RL com CBSS): Vitória

(a)

Jogadores (match)	Total de Ações Sugeridas pelas regras CBSS (em 3.000 jogos)		Total de Ações Sugeridas pela MLP (em 3.000 jogos)		Total de Ações Aleatórias (em 3.000 jogos)		Total de Casos (em 3.000 jogos)	
	ACE-RL (com CBSS)	ACE-RL (sem CBSS)	ACE-RL (com CBSS)	ACE-RL (sem CBSS)	ACE-RL (com CBSS)	ACE-RL (sem CBSS)	ACE-RL (com CBSS)	ACE-RL (sem CBSS)
ACE-RL com CBSS x ACE-RL sem CBSS	337	0	77.593	110.997	1.964	4.012	79.894	115.009

(b)

6.4.3 Avaliando a Escolha de Movimentos com Relação ao *Cake*

Os testes realizados nesta seção têm como objetivo estimar a taxa média de coincidência entre os 15 primeiros movimentos escolhidos pela versão *ACE-RL com CBSS* e seus oponentes, em jogos reais entre eles e os movimentos que, nas mesmas situações, seriam indicados pelo "*conselheiro de movimento*" do forte e bem sucedido agente supervisionado de Damas *Cake*. Em outras palavras, esses testes visam avaliar o quão perto o "raciocínio" desses agentes está com o "raciocínio" de *Cake*. É importante destacar que a versão do *Cake* utilizada nos experimentos faz uso de *opening book*, base de início de jogo com aproximadamente 2 milhões de movimentos de abertura. Os jogos utilizados nesta seção para avaliar a taxa média de movimentos coincidentes em relação ao *Cake* são os mesmos 12 jogos de torneio realizados entre as duas versões do sistema *ACE-RL-Checkers* e o agente *LS-VisionDraughts* apresentados nas seções 6.4.1 e 6.4.2. Este trabalho limitou

as análises dos *matches* desses jogos para os 15 primeiros movimentos devido aos seguintes fatos: primeiro, esses *matches* precisaram ser executados manualmente de forma que cada movimento realizado por cada agente pudesse ser comparado com o movimento que seria indicado por *Cake* na mesma situação; segundo, o objetivo desta seção é avaliar o desempenho do sistema *ACE-RL-Checkers* para além dos 10 primeiros movimentos, visto que a versão proposta neste capítulo faz uso da base de *regras CBSS*. Tabela 25 mostra os resultados obtidos nesses testes.

Tabela 25 – Taxa média de coincidência entre os movimentos escolhidos pelo *ACE-RL com CBSS* e seus oponentes, quando comparados com àqueles que seriam escolhidos pelo *Cake* na mesma situação.

Jogadores (Matches)	ACE-RL (com CBSS)	ACE-RL (sem CBSS)	LS-VisionDraughts
(primeiros 15 movimentos sobre os 12 jogos do torneio)	(Média de coincidência com Cake)	(Média de coincidência com Cake)	(Média de coincidência com Cake)
ACE-RL sem CBSS x LS-VisionDraughts	-	49,44%	42,22%
ACE-RL com CBSS x LS-VisionDraughts	66,11%	-	41,67%
ACE-RL com CBSS x ACE-RL sem CBSS	63,33%	46,11%	-

Como pode ser visto na tabela, a inclusão de uma base de *regras CBSS* na arquitetura do sistema *ACE-RL-Checkers* foi determinante para melhorar o nível de inteligência do agente nas fases iniciais do jogo, fato que proporcionou um aumento considerável na taxa média de movimentos coincidentes em relação ao *Cake*, chegando a uma média de 64,72%, se comparado com a média obtida por seus oponentes.

6.4.4 Análises Estatística de Wilcoxon

Esta seção tem como objetivo utilizar o método de estatística não-paramétrica *Wilcoxon* para validar estatisticamente o ganho real obtido pela versão *ACE-RL com CBSS*, proposta neste capítulo, nos testes realizados nas seções 6.4.2 e 6.4.3. Para tanto, de acordo com o teste de *Wilcoxon*, os seguintes passos devem ser seguidos a fim de provar que a *hipótese nula* H_0 – definida aqui como: a versão *ACE-RL com CBSS* possui o mesmo nível de desempenho de seus oponentes – deve ser rejeitada :

Passo I: Seja T a menor soma dos *ranks* (ordenações) positivas e negativas dentre as diferenças dos pares de amostras (resultados apresentados nas seções 6.4.2 e 6.4.3), isto é, $T = Min\{R^+, R^-\}$. Particularmente neste trabalho, R^+ representa a soma dos *ranks* em que *ACE-RL com CBSS* supera seus oponentes, enquanto R^- é a soma

dos ranks onde *ACE-RL com CBSS* é superado pelos seus oponentes. Use uma tabela de estatística apropriada ou ferramenta para determinar o *teste estatístico*, *valor crítico* ou *p-value* (dado provido pelo teste);

Passo II: Rejeita a *hipótese nula* H_0 se o *teste estatístico* $\leq$ *valor crítico* ou se *p-value* $\leq$ α (nível de significância). Quanto mais baixo é o valor de α, mais forte é a evidência contra a *hipótese nula* H_0.

Dessa forma, as tabelas 26-a) e 26-b) mostram os testes estatísticos de *Wilcoxon* correspondentes aos resultados obtidos na segunda e terceira análises apresentadas nas seções 6.4.2 e 6.4.3, respectivamente. Tabela 26-a) mostra os dados estatísticos de paridade extraídos a partir do torneio de 12 jogos realizados entre as versões *ACE-RL com CBSS* e *ACE-RL sem CBSS* na seção 6.4.2. Os valores R^+, R^- e *p-value* foram computados utilizando o *software* de estatística *SPSS*. Conforme pode ser visto na tabela 26-a), *ACE-RL com CBSS* é superior à sua versão predecessora, versão que não utiliza uma base de *regras CBSS*, com um nível de significância $\alpha = 0{,}04$. Tal fato rejeita a *hipótese nula*, o que indica que os resultados alcançados pelo *ACE-RL com CBSS*, nos jogos de torneio realizado na seção 6.4.2, são realmente diferenciados em relação àqueles obtidos pela versão *ACE-RL sem CBSS*.

Tabela 26 – a) Teste da ordenação dos sinais de *Wilcoxon* aplicados aos resultados da seção 6.4.2; b) Teste da ordenação dos sinais de *Wilcoxon* aplicados aos resultados da seção 6.4.3.

Torneio de Jogos			
Matches (comparações pareadas em 12 jogos)	**R+**	**R-**	***p-value***
ACE-RL com CBSS x ACE-RL sem CBSS	15,00	0,00	0,038

(a)

Média de movimentos coincidentes em relação ao Cake			
Matches (comparações pareadas em 12 jogos)	**R+**	**R-**	***p-value***
ACE-RL sem CBSS x LS-VisionDraughts	58,50	19,50	0,125
ACE-RL com CBSS x LS-VisionDraughts	78,00	0,00	0,002
ACE-RL com CBSS x ACE-RL sem CBSS	78,00	0,00	0,002

(b)

Analogamente, a tabela 26-b) mostra os valores R^+, R^- e *p-value* computados para todas as comparações pareadas que refletem a taxa média de coincidência entre os movimentos escolhidos pelo sistema *ACE-RL com CBSS* e seus oponentes, quando comparados com aqueles que seriam escolhidos pelo *Cake* na mesma situação. Essas taxas médias de coincidência foram obtidas nos 12 jogos de torneios executados na seção 6.4.3. Conforme apresentado na tabela 26-b), *ACE-RL com CBSS* é superior à sua versão predecessora e ao agente *LS-VisionDraughts* com um alto nível de significância $\alpha = 0{,}01$. Tal fato rejeita fortemente a *hipótese nula*, o que indica que os 15 movimentos iniciais executados pelo *ACE-RL com CBSS* são bem mais próximos daqueles indicados por *Cake*, quando comparados com os movimentos executados por seus oponentes. Já a versão *ACE-RL sem CBSS* é superior ao agente *LS-VisionDraughts* com um nível de significância $\alpha = 0{,}2$.

6.5 Considerações Relativas ao Capítulo

Este capítulo apresentou uma versão estendida do sistema *ACE-RL-Checkers* que além de introduzir flexibilidade de tomada de decisão através de um mecanismo que se adapta ao perfil de seu oponente no decorrer de um jogo, lida com a fragilidade do agente associada ao problema do *cold-start* nas fases iniciais do jogo de Damas, que é quando o agente nada sabe sobre o perfil de seu oponente. Para implementar tal arquitetura, o autor adotou uma versão probabilística da técnica *EAC* combinada com os conhecimentos provenientes de um agente estático, treinado por *AR*, e de uma base de *regras de experiência*, minerada a partir de registros de jogos de especialistas humanos. O desempenho do sistema proposto foi comparado com a melhor versão predecessora obtida no capítulo 5 e os resultados confirmam a melhora da versão que adota uma base de *regras CBSS* em relação a 3 aspectos: desempenho no início do jogo – medido através da taxa de movimentos iniciais coincidentes em relação ao grande jogador supervisionado *Cake*; tempo de treinamento; e melhora na acurácia dos casos gerados pela técnica *EAC* – medida através dos indicadores quantidade total de movimentos aleatórios gerados pela técnica *EAC* e a quantidade total de casos armazenados na *biblioteca*.

O próximo capítulo apresenta as principais contribuições deste trabalho, discute algumas limitações encontradas para o desenvolvimento desta pesquisa e os trabalhos futuros propostos.

Capítulo 7

Conclusões e Trabalhos Futuros

7.1 Considerações Finais

Este trabalho apresentou o sistema automático jogador de Damas híbrido *ACE-RL-Checkers* que combina as técnicas de aprendizagem de máquina *Elicitação Automática de Casos*, *Aprendizagem por Reforço* e *Mineração de Padrões Sequenciais*. Tal combinação permite ao agente ser dotado de um mecanismo dinâmico de tomada de decisões que se adapta ao perfil de seu oponente no decorrer de um jogo e lida com a fragilidade do agente associada ao problema do *cold-start* nas fases iniciais do jogo de Damas, que é quando o agente nada sabe sobre o perfil de seu oponente.

Nessa direção, primeiro foi construído a "identidade" do sistema *ACE-RL-Checkers*, isto é, o módulo tomador de decisões estático baseado em *Rede Neural de Perceptron Multicamadas* e treinado por *Aprendizagem por Reforço*. Tal módulo corresponde ao agente automático *LS-VisionDraughts* que combina *AG* com uma eficiente estratégia de busca baseada em algoritmo *Alfa-Beta*, tabela de transposição, aprofundamento iterativo e ordenação parcial da árvore de busca. Os resultados obtidos no capítulo 4 demonstram que apesar do *LS-VisionDraughts* ser bastante eficaz em suas tomadas de decisão, tal arquitetura apresenta o inconveniente de ser extremamente previsível, executando sempre o mesmo movimento diante de um mesmo tabuleiro e independente do adversário. Note que tal comportamento não permite ao agente *LS-VisionDraughts* evoluir seu nível de jogo, evitando alcançar tabuleiros de jogos desfavoráveis (que o leva a derrota), quando enfrenta adversários mais difíceis. Em outras palavras, *LS-VisionDraughts* não é capaz de evoluir seu nível de jogo observando sua própria experiência contra diferentes adversários. Neste sentido, com o objetivo de introduzir uma abordagem não determinística de tomada de decisão, na sequência, o presente trabalho reproduziu o agente automático *CHEBR* que é uma arquitetura jogadora de Damas baseada apenas em *Elicitação Automática de Casos*. Tal agente foi proposto por Powell e faz uso de uma abordagem probabilística que realiza exploração pseudo-aleatória no espaço de busca com o objetivo de aprender a jogar Damas automaticamente. Essas explorações pseudo-aleatórias permitem que o agente apresente

um comportamento extremamente adaptativo e não determinístico. Entretanto, conforme apresentado na seção 5.7.1, ao avaliar *CHEBR* contra *LS-VisionDraughts*, foi observado uma alta frequência de tomada de decisão aleatória por parte do agente *CHEBR*, fato que comprometeu seu desempenho nos torneios.

Considerando tais fatos, a arquitetura híbrida *ACE-RL-Checkers* foi proposta e implementada no capítulo 5, combinando primeiramente, as habilidades das abordagens de *AM Aprendizagem por Reforço* e *Elicitação Automática de Casos*, ao mesmo tempo em que elimina as suas fragilidades. Mais especificamente, com o conhecimento provido pela rede *MLP* do *LS-VisionDraughts* foi possível direcionar a exploração aleatória da *Elicitação Automática de Casos* para regiões mais promissoras no espaço de busca, fato que refinou a qualidade das tomadas de decisão e reduziu a quantidade de execução de movimentos aleatórios. Por outro lado, a nova dinâmica aleatória do *ACE-RL-Checkers* também introduziu adaptabilidade ao agente, uma vez que as tomadas de decisão passaram a não ser mais determinísticas e sim, baseadas na dinâmica corrente de jogadas de seus oponentes. Com tal abordagem híbrida, *ACE-RL-Checkers* superou os agentes *CHEBR* e *LS-VisionDraughts* em diversos *jogos de torneio* realizados.

Além disso, também foram investigadas, no capítulo 5, duas novas estratégias alternativas para calcular o valor do *rating* dos casos gerados no contexto do *ACE-RL-Checkers*. A estratégia *Memória Positiva Geral* mostrou ser superior às estratégias de *Decaimento de Memória* e *Memória de Confiança Superior*. Tal estratégia contribuiu para reduzir significantemente o percentual de *novas ações aleatórias* geradas pela técnica *EAC*, bem como a quantidade total de casos armazenados na *biblioteca* do *SRBC*.

Por outro lado, ao conceber a arquitetura híbrida baseada em *AR* e *EAC*, é necessário evitar a seguinte fragilidade: nas fases iniciais do jogo em que a quantidade de casos disponíveis na *biblioteca* da técnica *EAC* é extremamente baixa em função do exíguo conhecimento do perfil do adversário, o desempenho do agente é geralmente comprometido pela alta frequência de execução de movimentos aleatórios sobre o tabuleiro. Tal fragilidade ocorre devido a dois motivos: primeiro, porque a *biblioteca de casos* é sempre inicializada (zerada) para cada oponente com o qual o agente interage, isto é, o agente começa jogando sem nenhum conhecimento sobre seu adversário; segundo, devido às características inerentes à técnica *EAC*, a tomada de decisão dinâmica do agente é guiada, ora pelo perfil do adversário – que é quando *casos* são recuperados da *biblioteca*, ora aleatoriamente – que é quando a técnica *EAC* não recupera nenhum caso da *biblioteca*. Essa última situação ocorre em função do próprio mecanismo *EAC* de seleção pseudo-aleatória de casos optar por explorar novas regiões no espaço de busca ou em função da *biblioteca* não possuir informações suficientes reunidas sobre um determinado perfil de jogo do adversário (situação em que há escassez de casos). Visando atacar tal fragilidade, no capítulo 6 foi implementado uma versão estendida que incorpora na arquitetura do *ACE-RL-Checkers* um novo módulo baseado na técnica de *Mineração de Padrões Sequenciais*. Tal módulo

é responsável por gerar uma base de *regras de experiência*, minerada a partir de registros de jogos de torneio contendo sequências de movimentos de especialistas humanos, com o objetivo de prover conhecimento de experiência humana à dinâmica pseudo-aleatória da técnica *EAC* de forma a acelerar o processo de adaptação do agente ao perfil de seu adversário nas fases iniciais do jogo. Os resultados apresentados na seção 6.4 demonstram que a versão estendida é superior à melhor versão predecessora obtida no capítulo 5 em relação a 3 aspectos: desempenho no início do jogo, tempo de treinamento e melhora na acurácia dos casos gerados pela técnica *EAC*.

7.2 Contribuições Científicas

Como principais contribuições obtidas neste trabalho, podem-se destacar:

1. A obtenção de uma eficiente plataforma jogadora de Damas baseada apenas em *Aprendizagem por Reforço*: o *LS-VisionDraughts*;

2. A obtenção de uma nova abordagem híbrida não supervisionada que combina as técnicas de aprendizagem de máquina *AR* e *EAC* para ser aplicada em agentes jogadores automáticos. Tal abordagem utiliza o conhecimento provido por um agente baseado em *AR* para introduzir um certo controle na exploração aleatória da técnica *EAC*, de forma a direcionar a escolha de movimentos para regiões mais promissoras no espaço de busca, fato que refina a qualidade das tomadas de decisão do agente. Além disso, como segunda contribuição, a nova dinâmica aleatória da abordagem híbrida norteada por conhecimento introduz adaptabilidade ao agente, uma vez que as tomadas de decisão não serão mais determinísticas e as escolhas de movimento do agente serão baseadas na dinâmica corrente de cada jogo. Em tal dinâmica, o perfil de jogadas do oponente é traçado automaticamente ao longo das partidas disputadas;

3. A obtenção de novas estratégias para cálculo do *rating* de forma a melhorar a acurácia dos casos gerados no contexto da técnica *EAC*, fato que permite reduzir a execução de movimentos aleatórios e a quantidade de casos armazenados em memória;

4. A obtenção de uma nova abordagem baseada em *MPS* para tratar o problema do *cold-start* em agentes que são dotados da habilidade de, progressivamente, traçar um perfil de seu adversário ao longo dos jogos. Particularmente, nas fases iniciais do jogo em que as informações referentes a esse perfil é ainda escasso, a nova abordagem utiliza *regras de experiência* para prover conhecimento de experiência humana à dinâmica pseudo-aleatória da técnica *EAC* de forma a acelerar o processo de adaptação do agente ao perfil de seu adversário. Tais regras são extraídas

a partir de registros de jogos contendo sequências de movimentos de especialistas humanos.

É importante destacar que as metodologias envolvidas nessas contribuições podem ser estendidas para outros jogos de soma zero, isto é, jogos em que o ganho (ou perda) de um dos jogadores é equilibrado, exatamente, pela(s) perda(s), ou ganho(s), do(s) outro(s) participante(s), de tal modo que a soma do total de ganhos dos participantes subtraída de todas as suas perdas será zero (para mais detalhes sobre jogos de soma zero, veja [70]). Entretanto, para que tais metodologias possam ser aplicadas integralmente a outros jogos, a seguinte restrição deve ser avaliada: o *AG* utilizado na construção do agente *LS-VisionDraughts* tem como objetivo apenas selecionar, automaticamente, *features* que possam representar o tabuleiro de Damas na entrada de uma rede *MLP*, ao invés de propor ou criar novas *features* que representam o domínio do jogo. Neste sentido, para que o *AG* possa ser utilizado conforme metodologia proposta no objetivo 1 da seção 1.3.1, é necessário que os jogos de soma zero já tenham, prédefinidas, todas as *features* que representam o domínio (estado) desses jogos.

7.3 Limitações

Algumas das limitações deste trabalho são descritas a seguir:

- Devido à restrição de memória do *desktop* utilizado nos experimentos e pelo uso de tabelas de transposição, tanto no algoritmo de busca *alfa-beta* quanto no algoritmo *EAC*, as *regras de experiência* tiveram que ser armazenadas em arquivos comuns. Entretanto, tal limitação pouco influenciou no tempo de consulta das *regras* por serem utilizadas apenas na fase inicial do jogo de Damas;

- Inicialmente o autor tentou entrar em contato com as federações internacionais de Damas, *ACF* e *WCDF*, para tentar segregar os registros de jogos humanos por nível macro de habilidades, tais como, iniciante, básico, intermediário, avançado e especialista, de acordo com o *score* de pontuação atualizado, periodicamente, no *ranking* dessas federações. Entretanto, os responsáveis pelas federações *ACF* e *WCDF* não colaboraram com o objetivo proposto e o autor teve que contar com a ajuda de Fierz, autor do *Cake*, para coletar jogos apenas de especialistas de Damas – a maioria desses jogos foram extraídos da base *OCA* (*Open Checkers Archive*).

7.4 Produção Bibliográfica

No que tange a produção científica relacionada com esta tese, foram publicados os seguintes artigos em eventos internacionais:

1. *Artigo de revista referente ao primeiro objetivo da seção 1.3.1*: "NETO, H. C. et al. *LS-VisionDraughts: Improving the Performance of an Agent for Checkers by Integrating Computational Intelligence, Reinforcement Learning and a Powerful Search Method*. Journal of Applied Intelligence (APIN), Springer US, v. 41. n. 2, p. 525-550, 2014". (Ciência da Computação: Qualis B1);

2. *Artigo completo em conferência referente ao segundo e terceiro objetivos da seção 1.3.1*: "NETO, H. C.; JULIA, R. M. S. *ACE-RL-Checkers: Improving Automatic Case Elicitation Through Knowledge Obtained by Reinforcement Learning in Player Agents*. In: 2015 IEEE Conference on Computational Intelligence and Games (CIG). p. 328–335, Tainan, Taiwan, 2015". (Ciência da Computação: Qualis B1);

3. *Artigo completo em conferência referente ao quarto objetivo da seção 1.3.1*: "NETO, H. C.; JULIA, R. M. S.; DUARTE, V. A. R. *Improving the Accuracy of the Cases in the Automatic Case Elicitation-Based Hybrid Agents for Checkers*. In: 2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI), Vietri Sul Mare, Italy, p. 912–919, 2015". (Ciência da Computação: Qualis A2).

O artigo listado abaixo, referente ao quinto objetivo da seção 1.3.1, foi submetido a um periódico e está em processo de avaliação:

1. "NETO, H. C.; JULIA, R. M. S. *ACE-RL-Checkers: Introducing Decision-making Adaptability Through the Integration of Automatic Case Elicition, Reinforcement Learning and Mining Sequential Patterns*. Journal of Knowledge and Information Systems (KAIS)". (Ciência da Computação: Qualis B1).

7.5 Trabalhos Futuros

Durante o desenvolvimento desta pesquisa, algumas questões foram levantadas para melhorar o desempenho da metodologia proposta. Entretanto, estas questões não foram investigadas no escopo desta pesquisa, constituindo assim, assuntos para trabalhos futuros. A seguir são listadas as questões levantadas que proporcionarão trabalhos futuros:

- ❑ Substituir a arquitetura da *MLP* adotada pelo agente *LS-VisionDraughs* por uma *rede neural convolucional profunda* que seja capaz de construir seu próprio mapa de *características* do tabuleiro de Damas com o objetivo de eliminar o conjunto de *features* do mapeamento *NET-FEATUREMAP*;

- ❑ Investigar novas formas de integrar e balancear a natureza pseudo-aleatória da técnica *EAC* com o dilema *exploration* e *exploitation* inerente à estratégia *UCM* proposta na seção 5.5.3;

- ❏ Segregar os registros de jogos oficiais de Damas, ministrados pelas federações *ACF* e *WCDF*, para diferentes níveis macro de habilidades, tais como, iniciante, básico, intermediário, avançado e especialista, de forma a gerar várias bases de *regras de experiência* para cada especialidade. Um estudo deverá ser realizado para agrupar os diferentes *scores* (pontuações) dos jogadores no *ranking* das federações *ACF* e *WCDF* nesses diferentes níveis de habilidade aqui propostos. O objetivo é incluir uma nova funcionalidade no agente *ACE-RL-Checkers* para identificar o nível de habilidade inicial de seu oponente com base nos primeiros movimentos executados por esse e comparando-os com o conhecimento armazenado em suas diferentes bases de *regras de experiência*. A partir de então, uma determinada base de *regras de experiência* será escolhida e automaticamente refinada a medida que o agente for jogando contra o seu oponente utilizando a técnica híbrida que combina *AR* e *EAC*;

- ❏ Propor uma estratégia de refinamento de casos gerados pela técnica *EAC* de forma a eliminar casos "obsoletos" da *biblioteca* que não são mais úteis ao agente com base em suas últimas experiências vivenciadas.

Referências

1 PANCERI, S. S. **SIIEE - Sistema de Raciocínio Baseado em Casos para Análise de Perfil de Candidatos a Vagas de Estágio**. [S.l.], 2012. Trabalho de conclusão do curso de bacharel em Ciência da Computação.

2 WANG, L.; WANG, Y.; LI, Y. Mining Experiential Patterns from Game-logs of Board Game. **International Journal of Computer Games Technology**, Hindawi Publishing Corp., v. 2015, 2015. ISSN 1687-7047. Disponível em: <http://dx.doi.org/10.1155/2015/576201>.

3 MATSUBARA, H.; IIDA, H.; GRIMBERGEN, R. **Chess, Shogi, Go, natural developments in game research**. 1997.

4 HERIK, H. J. V.; UITERWIJK, J. W. H. M.; RIJSWIJCK, J. V. Games solved: Now and in the future. **Artificial Intelligence**, v. 134, p. 277–311, 2002.

5 CAMPOS, P.; LANGLOIS, T. Abalearn: Efficient Self-Play Learning of the game Abalone. In: **INESC-ID, Neural Networks and Signal Processing Group**. [S.l.: s.n.], 2003.

6 LYNCH, M. **An Application of Temporal Difference Learning to Draughts**. Dissertação (Mestrado) — University of Limerick, Ireland, 1997.

7 MACHADO, M. C.; FANTINI, E. P.; CHAIMOWICZ, L. Player Modeling: Towards a Common Taxonomy. In: **16th International Conference on Computer Games**. [S.l.: s.n.], 2011. p. 50–57.

8 NEUMANN, J. V.; MORGENSTERN, O. **Theory of Games and Economic Behavior**. [S.l.]: Princeton University Press, 1944.

9 SHANNON, C. E. Programming a Computer for Playing Chess. **Philosophical Magazine**, n. 314, p. 256–275, 1950.

10 MARSLAND, T. The Anatomy of Chess Programs. **Papers from the 1997 AAAI Work-shop**, 1997.

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

12 NETO, H. C.; JULIA, R. M. S. LS-Draughts - A Draughts Learning System Based on Genetic Algorithms, Neural Network and Temporal Differences. **IEEE Congress on Evolutionary Computation**, p. 2523–2529, 2007.

13 SAMUEL, A. L. Some Studies in Machine Learning Using the Game of Checkers. **IBM Journal of Research and Development**, p. 211–229, 1959.

14 _____. Some Studies in Machine Learning Using the Game of Checkers ii - Recent Progress. **IBM Journal of Research and Development**, p. 601–617, 1967.

15 SCHAEFFER, J. et al. CHINOOK: The World Man-Machine Checkers Champion. **AI Magazine**, v. 17, n. 1, p. 21–30, 1996.

16 _____. Checkers is Solved. **Science Express**, v. 328, n. 5844, p. 1518, 2007.

17 FOGEL, D. B.; CHELLAPILLA, K. Verifying Anaconda's Expert Rating by Competing Against Chinook: Experiments in Co-Evolving a Neural Checkers Player. **Neurocomputing**, v. 42, n. 1-4, p. 69–86, 2001.

18 FOGEL, D. B. **Blondie24: Playing at the Edge of AI**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

19 SCHAEFFER, J.; HLYNKA, M.; JUSSILA, V. Temporal Difference Learning Applied to a High Performance Game-Playing Program. **International Joint Conference on Artificial Intelligence**, p. 529–534, 2001.

20 POWELL, J. H.; HAUFF, B. M.; HASTINGS, J. D. Utilizing Case-Based Reasoning and Automatic Case Elicitation to Develop a Self-Taught Knowledgeable Agent. In: **Challenges in Game Artificial Intelligence: Papers from the AAAI Workshop (Technical Report WS-0404)**. [S.l.]: AAAI Press, 2004.

21 _____. Evaluating the Effectiveness of Exploration and Accumulated Experience in Automatic Case Elicitation. In: **In Proceedings of ICCBR 2005**. [S.l.]: Springer, 2005. p. 397–407.

22 CHEHELTANI, S. H.; EBADZADEH, M. M. Immune based fuzzy agent plays checkers game. **Applied Soft Computing**, v. 12, n. 8, p. 2227–2236, 2012.

23 AL-KHATEEB, B.; KENDALL, G. Effect of Look-Ahead Depth in Evolutionary Checkers. **Journal of Computer Science and Technology**, v. 27, n. 5, p. 996–1006, 2012.

24 DUARTE, V. A. R.; JULIA, R. M. S. MP-Draughts: Ordering the Search Tree and Refining the Game Board Representation to Improve a Multi-agent System for Draughts. In: **2012 IEEE 24th International Conference on Tools with Artificial Intelligence (ICTAI)**. [S.l.: s.n.], 2012. v. 1, p. 1120–1125.

25 AL-KHATEEB, B.; KENDALL, G. Introducing Individual and Social Learning Into Evolutionary Checkers. **IEEE Transactions on Computational Intelligence and AI in Games**, p. 258–269, 2012.

26 TOMAZ, L. B. P.; JULIA, R. M. S.; BARCELOS, A. R. A. Improving the accomplishment of a neural network based agent for draughts that operates in a distributed learning environment. In: **2013 IEEE 14th International Conference on Information Reuse Integration (IRI)**. [S.l.]: IEEE, 2013. p. 262–269.

27 NETO, H. C. et al. LS-VisionDraughts: improving the performance of an agent for checkers by integrating computational intelligence, reinforcement learning and a powerful search method. **Applied Intelligence**, Springer US, v. 41, n. 2, p. 525–550, 2014. ISSN 0924-669X.

28 ELNAGGAR, A. A. et al. Autonomous checkers robot using enhanced massive parallel game tree search. In: **Informatics and Systems (INFOS), 2014 9th International Conference on**. [S.l.: s.n.], 2014. p. 35–44.

29 ZHAO, Z. et al. The Game Method of Checkers based on Alpha-Beta Search Strategy with Iterative Deepening. In: **The 26th Chinese Control and Decision Conference (2014 CCDC)**. [S.l.: s.n.], 2014. p. 3371–3374.

30 FRANKLAND, C.; PILLAY, N. Evolving Heuristic Based Game Playing Strategies for Checkers Incorporating Reinforcement Learning. In: _____. **Advances in Nature and Biologically Inspired Computing: Proceedings of the 7th World Congress on Nature and Biologically Inspired Computing (NaBIC2015) in Pietermaritzburg, South Africa**. [S.l.]: Springer International Publishing, 2015. p. 165–178.

31 NETO, H. C.; JULIA, R. M. S. ACE-RL-Checkers: Improving automatic case elicitation through knowledge obtained by reinforcement learning in player agents. In: **2015 IEEE Conference on Computational Intelligence and Games (CIG)**. [S.l.: s.n.], 2015. p. 328–335.

32 DUARTE, V. A. R. et al. MP-Draughts: Unsupervised Learning Multi-agent System Based on MLP and Adaptive Neural Networks. In: **2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)**. [S.l.: s.n.], 2015. p. 920–927.

33 NETO, H. C.; JULIA, R. M. S.; DUARTE, V. A. R. Improving the Accuracy of the Cases in the Automatic Case Elicitation-Based Hybrid Agents for Checkers. In: **2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)**. [S.l.: s.n.], 2015. p. 912–919.

34 DUARTE, V. A. R.; JULIA, R. M. S. Improving NetFeatureMap-based Representation through Frequent Pattern Mining in a Specialized Database. In: **2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)**. [S.l.: s.n.], 2016.

35 FIERZ, M. C. **Cake Informations**. Morgan kaufmann. [S.l.], 2008. Available in: http://www.fierz.ch/cake.php.

36 _____. **CheckerBoard Program - version 1.72**. [S.l.], 2011. Available in: http://www.fierz.ch/checkerboard.php and http://www.fierz.ch/engines.php.

37 CAIXETA, G. S.; JULIA, R. M. S. A Draughts Learning System based on Neural Networks and Temporal Differences: The Impact of an Efficent Tree-Search Algorithm. **The 19th Brazilian Symposium on Artificial Intelligence, SBIA**, 2008.

38 DUARTE, V. A. R.; JULIA, R. M. S. MP-Draughts: A Multiagent Reiforcement Learning System Based on MPL and Kohonen-SOM Neural Networks. **IEEE Internation Conference on Systems, Man and Cybernetics**, p. 2270–2275, 2009.

39 BARCELOS, A. R. A.; JULIA, R. M. S.; JR., R. M. 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.

40 LYNCH, M.; GRIFFITH, N. NeuroDraughts: The Role of Representation, Search, Training Regime and Architecture in a TD Draughts Player. **Eighth Ireland Conference on Artificial Intelligence**, Ireland, p. 67–72, 1997. Disponível em: <$http://iamlynch.com/nd.html$>.

41 BARD, N.; BOWLING, M. Particle Filtering for Dynamic Agent Modelling in Simplified Poker. In: **Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 1**. [S.l.: s.n.], 2007. (AAAI'07), p. 515–521.

42 GóMEZ-SEBASTIà, I. et al. A Flexible Agent-Oriented Solution to Model Organisational and Normative Requirements in Assistive Technologies. In: **Proceedings of the 2010 Conference on Artificial Intelligence Research and Development: Proceedings of the 13th International Conference of the Catalan Association for Artificial Intelligence**. [S.l.]: IOS Press, 2010. p. 79–88.

43 DIA, H. An agent-based approach to modelling driver route choice behaviour under the influence of real-time information. **Transportation Research Part C: Emerging Technologies**, v. 10, p. 331–349, 2002.

44 BAZGHANDI, A. Techniques, Advantages and Problems of Agent Based Modeling for Traffic Simulation. In: **IJCSI - International Journal of Computer Science**. [S.l.: s.n.], 2012. p. 115–119.

45 SERRANO, E. et al. Strategies for avoiding preference profiling in agent-based e-commerce environments. **Applied Intelligence**, Springer US, v. 40, n. 1, p. 127–142, 2014.

46 SMYTH, B. Case-Based Recommendation. In: **The Adaptive Web**. [S.l.]: Springer Berlin Heidelberg, 2007, (Lecture Notes in Computer Science, v. 4321). p. 342–376.

47 GARRIDO, A.; MORALES, L.; SERINA, I. Applying Case-Based Planning to Personalized E-learning. In: **DMS**. [S.l.]: Knowledge Systems Institute, 2011. p. 228–233.

48 _____. Using AI Planning to Enhance E-Learning Processes. In: **ICAPS**. [S.l.: s.n.], 2012.

49 GARRIDO, A. et al. On the automatic compilation of e-learning models to planning. **Knowledge Eng. Review**, v. 28, n. 2, p. 121–136, 2013.

50 PEDERSEN, C.; TOGELIUS, J.; YANNAKAKIS, G. N. Modeling Player Experience for Content Creation. **Computational Intelligence and AI in Games, IEEE Transactions**, v. 2, n. 1, p. 54–67, 2010.

51 AVONTUUR, T.; SPRONCK, P.; ZAANEN, M. van. Player Skill Modeling in Starcraft II. In: **AIIDE**. [S.l.: s.n.], 2013.

52 COWLEY, B. U.; CHARLES, D. Short Literature Review for a General Player Model Based on Behavlets. **CoRR**, abs/1603.06996, 2016.

53 COWLEY, B. U. How to advance general game playing artificial intelligence by player modelling. **CoRR**, abs/1606.00401, 2016. Disponível em: <http://arxiv.org/abs/1606.00401>.

54 HE, H. et al. Opponent Modeling in Deep Reinforcement Learning. **Proceedings of the 33rd International Conference on Machine Learning (ICML 16)**, v. 48, June 2016.

55 CHEN, Z.; YI, D. **The Game Imitation: A Portable Deep Learning Model for Modern Gaming AI**. [S.l.], 2016. Available in: http://cs231n.stanford.edu/reports2016/113Report.pdf.

56 LEARNING to Predict Life and Death from Go Game Records. In: . [S.l.: s.n.], 2005. v. 175, p. 258–272. Heuristic Search and Computer Game Playing IV.

57 WENDER, S. **Data Mining and Machine Learning with Computer Game Logs**. [S.l.]. Project Report available in: https://www.cs.auckland.ac.nz/research/gameai/projects.

58 ESAKI, T.; HASHIYAMA, T. Extracting Human Players Shogi Game Strategies from Game Records using Growing SOM. In: **2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)**. [S.l.: s.n.], 2008. p. 2176–2181.

59 WEBER, B. G.; MATEAS, M. A Data Mining Approach to Strategy Prediction. In: **2009 IEEE Symposium on Computational Intelligence and Games**. [S.l.: s.n.], 2009. p. 140–147.

60 TAKEUCHI, S.; KANEKO, T.; YAMAGUCHI, K. Evaluation of Game Tree Search Methods by Game Records. **IEEE Transactions on Computational Intelligence and AI in Games**, v. 2, n. 4, p. 288–302, Dec 2010.

61 CORPORATION, I. **IBM Deep Blue**. [S.l.], 1997. Available in: http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue.

62 SINCLAIR, D. Using Example-Based Reasoning for Selective Move Generation in Two Player Adversarial Games. In: **Proceedings of the 4th European Workshop on Advances in Case-Based Reasoning**. [S.l.]: Springer-Verlag, 1998. (EWCBR '98), p. 126–135.

63 FLINTER, S.; KEANE, M. T. On the Automatic Generation of Case Libraries by Chunking Chess Games. In: **Proceedings of the 1st International Conference on Case Based Reasoning (ICCBR-95**. [S.l.: s.n.], 1995. p. 421–430.

64 SILVER, D. et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. **Nature**, v. 529, n. 7587, p. 484–489, 2016.

65 SCHEIN, A. I. et al. Methods and Metrics for Cold-start Recommendations. In: **Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval**. New York, NY, USA: ACM, 2002. (SIGIR '02), p. 253–260.

66 ADOMAVICIUS, G.; TUZHILIN, A. Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. **IEEE Transactions on Knowledge and Data Engineering**, v. 17, n. 6, p. 734–749, 2005.

67 MISIUNAS, T. **Realtime recommendation system for online games**. Dissertação (Mestrado) — School of Informatics - University of Edinburgh, Edinburgh, United Kingdom, 2014.

68 BANKS, S.; RAFTER, R.; SMYTH, B. The Recommendation Game: Using a Game-with-a-Purpose to Generate Recommendation Data. In: **Proceedings of the 9th ACM Conference on Recommender Systems**. New York, NY, USA: ACM, 2015. p. 305–308.

69 MüLLER, M.; ENZENBERGER, M. **Fuego - An Open-source Framework for Board Games and Go Engine Based on Monte-Carlo Tree Search**. [S.l.], 2009.

70 DUTTA, P. K. **Strategies and games: theory and practice**. Cambridge: MIT Press, 1999.

71 RUSSELL, S.; NORVIG, P. **Inteligência Artificial - Uma Abordagem Moderna (2a edição)**. [S.l.]: Editora Campus, 2004.

72 PLAAT, A. **Research Re: Search & Re-search**. Tese (Doutorado) — Tinbergen Institute and Department of Computer Science - Erasmus University, Rotterdam, The Netherlands, 1996.

73 HAYKIN, S. **Redes Neurais: Princípios e Prática (2º edição)**. Porto Alegre, RS: Bookman Editora, 2001.

74 MCCULLOCH, W.; PITTS, W. A Logical Calculus of the Ideas Immanent in Nervous Activity. **Bulletin of Mathematical Biophysics**, v. 5, p. 115–133, 1943.

75 CARBONELL, J. G.; MICHALSKI, R. S.; MITCHELL, T. M. Machine learning: A historical and methodological analysis. **AI Magazine**, p. 69–78, 1983. Disponível em: <https://pdfs.semanticscholar.org/3522/d171d0af99fb1e0a06f8d31734987967870a.pdf>.

76 CAIXETA, G. S. **VisionDraughts - Um Sistema de Aprendizagem de Jogos de Damas Baseado em Redes Neurais, Diferenças Temporais, Algoritmos Eficientes de Busca em Árvores e Informações Perfeitas Contidas em Bases de Dados**. Dissertação (Mestrado) — Faculdade de Computação - Universidade Federal de Uberlândia, Uberlândia, Brasil, 2008.

77 SCHAEFFER, J. **One Jump Ahead: Computer Perfection at Checkers (2nd edition)**. [S.l.]: Springer US, 2009. ISBN 9780387765761.

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

79 SUTTON, R. S. Learning to Predict by the Methods of Temporal Differences. **Machine Learning**, v. 3, n. 1, p. 9–44, 1988.

80 EIBEN, A. E.; SMITH, J. E. **Introduction to Evolutionary Computing**. [S.l.]: SpringerVerlag, 2003. ISBN 3540401849.

81 HOLLAND, J. H. **Adaptation in Natural and Artificial Systems**. [S.l.]: University of Michigan Press, 1975.

82 BENTLEY, P. J. **Digital Biology: How Nature Is Transforming Our Technology and Our Lives**. New York: Simon & Schuster Inc, 2002.

83 AAMODT, A.; PLAZA, E. Case-based reasoning; Foundational issues, methodological variations, and system approaches. **AI COMMUNICATIONS**, v. 7, n. 1, p. 39–59, 1994.

84 SRIKANT, R.; AGRAWAL, R. Mining Sequential Patterns: Generalizations and Performance Improvements. In: **Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology**. London, UK, UK: Springer-Verlag, 1996. p. 3–17.

85 MABROUKEH, N. R.; EZEIFE, C. I. A Taxonomy of Sequential Pattern Mining Algorithms. **ACM Computing Surveys**, v. 43, n. 1, p. 1–41, 2010.

86 SLIMANI, T.; LAZZEZ, A. Sequential Mining: Patterns and Algorithms Analysis. **International Journal of Computer and Electronics Research**, v. 2, p. 639–647, 2013.

87 HAN, J.; KAMBER, M.; PEI, J. **Data Mining: Concepts and Techniques**. 3ª. ed. [S.l.]: The Morgan Kaufmann Series in Data Management Systems, 2011. ISBN 9780123814791.

88 ZAKI, M. J. SPADE: An Efficient Algorithm for Mining Frequent Sequences. **Machine Learning**, Kluwer Academic Publishers, Hingham, MA, USA, v. 42, n. 1-2, p. 31–60, jan. 2001. ISSN 0885-6125.

89 YAN, X.; HAN, J.; AFSHAR, R. CloSpan: Mining Closed Sequential Patterns in Large Datasets. In: **Proceedings of the 3rd SIAM**. [S.l.: s.n.], 2003. p. 166–177.

90 PEI, J. et al. Mining sequential patterns by pattern-growth: the PrefixSpan approach. **IEEE Transactions on Knowledge and Data Engineering**, v. 16, n. 11, p. 1424–1440, Nov 2004.

91 LIN, M.-Y.; LEE, S.-Y. Fast Discovery of Sequential Patterns by Memory Indexing. **Data Warehousing and Knowledge Discovery**, Lecture Notes in Computer Science, Springer, Berlin, Germany, v. 2454, p. 150–160, 2002.

92 NETO, H. C.; JULIA, R. M. S.; CAIXETA, G. S. Theory and novel application of machine learning. In: _____. [S.l.]: I-Tech Education and Publishing, 2009. cap. LS-Draughts: Using Databases to Treat Endgame Loop in a Hybrid Evolutionary Learning System.

93 JONG, K. A. D.; SCHULTZ, A. C. Using Experience-Based Learning in Game Playing. **Fifth International Machine Learning Conference**, p. 284–290, 1988.

94 CHELLAPILLA, K.; FOGEL, D. B. 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**, California, USA, p. 857–863, 2000.

95 TESAURO, G. TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play. **Neural Computation**, MIT Press, v. 6, n. 2, p. 215–219, 1994.

96 _____. Temporal Difference Learning and TD-Gammon. **Communications of the ACM**, v. 38, n. 3, p. 58–68, 1995.

97 FOGEL, D. B. et al. A Self-Learning Evolutionary Chess Program. **Proceedings of the IEEE**, v. 92, n. 12, p. 1947–1954, 2004.

98 DARWEN, P. J. Why Co-Evolution Beats Temporal Difference Learning at Backgammon for a Linear Architecture, but not a Non-Linear Architecture. In: **Proceedings of the 2001 Congress on Evolutionary Computation CEC2001**. [S.l.]: IEEE Press, 2001. p. 1003–1010.

99 MADDISON, C. et al. Move Evaluation in Go Using Deep Convolutional Neural Networks. **International Conference on Learning Representations (ICLR 15)**, 2015.

100 SINGH, S. P.; SUTTON, R. S. Reinforcement Learning with Replacing Eligibility Traces. **Machine Learning**, v. 22, n. 1, p. 123–158, 1996.

101 MILLINGTON, I. **Artificial Intelligence for Games**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

102 ZOBRIST, A. L. **A Hashing Method with Applications for Game Playing**. Computer sciences department, university of wisconsin. [S.l.], 1970. Available in: https://minds.wisconsin.edu/handle/1793/57624.

103 STEVANOVIC, R. **Quantum Random Bit Generator Service**. [S.l.], 2007. Available in: http://random.irb.hr.

104 STIPCEVIC, M.; ROGINA, B. M. Quantum random number generator based on photonic emission in semiconductors. **Review of Scientific Instruments**, v. 78, n. 4, 2007.

105 BREUKER, D.; UITERWIJK, J.; HERIK, H. **Replacement Schemes for Transposition Tables**. [S.l.], 1994. Available in: http://citeseer.ist.psu.edu/112066.html.

106 SLATE, D. J.; ATKIN, L. R. **Chess Skill in Man and Machine**. [S.l.]: Springer-Verlag, 1977.

107 DERRAC, J. et al. A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. **Swarm and Evolutionary Computation**, v. 1, n. 1, p. 3–18, 2011.

108 AMERICAN Checkers Federation (ACF). [S.l.], 2014. Available in: http://www.usacheckers.com/.

109 WORLD Chechers and Draughts Federation (WCDF). [S.l.], 2014. Available in: http://www.wcdf.net/.