ADABA: uma nova abordagem de distribuição do Alfa-Beta - aplicação ao domínio do jogo de Damas

Lídia Bononi Paiva Tomaz

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

Uberlândia
2018

Lídia Bononi Paiva Tomaz

ADABA: uma nova abordagem de distribuição do Alfa-Beta - aplicação ao domínio do jogo 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: Rita Maria da Silva Julia

Uberlândia
2018

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

T655a 2018

Tomaz, Lídia Bononi Paiva, 1988-
ADABA [recurso eletrônico] : uma nova abordagem de distribuição do Alfa-Beta - aplicação ao domínio do jogo de Damas / Lídia Bononi Paiva Tomaz. - 2018.

Orientadora: Rita Maria da Silva Julia.
Tese (Doutorado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://dx.doi.org/10.14393/ufu.te.2019.316
Inclui bibliografia.
Inclui ilustrações.

1. Computação. 2. Jogo de damas por computador. 3. Algoritmos de computador. I. Julia, Rita Maria da Silva, 1960- (Orient.) II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3

Maria Salete de Freitas Pinheiro - CRB6/1262

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 "**ADABA: uma nova abordagem de distribuição do Alfa-Beta - aplicação ao domínio do jogo de Damas**" por **Lídia Bononi Paiva Tomaz** como parte dos requisitos exigidos para a obtenção do título de **Doutora em Ciência da Computação**.

Uberlândia, 17 de dezembro de 2018.

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

Banca Examinadora:

_____
Prof. Dr. Fabio Gagliardi Cozman
Universidade de São Paulo

_____
Prof. Dr. Jomi Fred Hubner
Universidade Federal de Santa Catarina

_____
Profa. Dra. Elaine Ribeiro de Faria
Universidade Federal de Uberlândia

_____
Prof. Dr. Paulo Henrique Ribeiro Gabriel
Universidade Federal de Uberlândia

Este trabalho é dedicado à minha querida família que não mediu esforços durante toda esta caminhada em me apoiar, segurar no braço, me levantar e seguir comigo até aqui.

Agradecimentos

Primeiramente agradeço a Deus por toda a sua criação e oportunidade a nós concedida para desenvolvermos a inteligência em prol de um mundo melhor.

Aos meus pais Valter Paiva e Fátima F. Bononi pela paciência e dedicação ofertadas desde meus primeiros dias de vida e por serem a base de força e moral que permitiram me tornar a pessoa que sou hoje.

Às minhas irmãs Alcione e Lívia pelo companheirismo que se fortaleceu com todas as mudanças que a vida trilhou para nós.

Aos meus cunhados Frederico Miranda e Edson Cordeiro Jr. pelo convívio e apoio, em especial, nos momentos difíceis.

Ao meu marido Guilherme e ao meu filhinho Artur que foram bençãos entregues a mim durante este percurso e se tornaram fundamentais para meu amadurecimento como pessoa. Com vocês, chegar até aqui tem um gosto mais especial!

À Profa. Dra. Rita pela excelente orientação na condução deste trabalho, pela confiança, conselhos, motivação, compreensão e parceria. Minha eterna admiração!

Ao Erisvaldo por tornar minha vida no programa de pós-graduação menos burocrática sempre repassando com boa vontade todas as informações necessárias e pela amizade construída ao longo deste caminho.

Ao Matheus Prandini por me deixar fazer parte de sua formação na graduação e contribuir tanto na condução deste trabalho.

Aos professores do programa de pós-graduação da Faculdade de Computação da UFU por partilharem experiências e conhecimentos.

Aos amigos e colegas que fiz no decorrer do doutorado, que compartilharam comigo direta e indiretamente este período de muito trabalho, com destaque à Valquíria Duarte, Henrique Castro, Clarimundo Júnior, Sara Melo e Aryadne Guardieiro.

Aos meus amigos e colegas de trabalho do IFTM Campus Patrocínio e Campus Avançado Uberaba Parque Tecnológico que estiveram presentes neste percurso.

A todos, de coração, muito obrigada!

"A persistência é o menor caminho do êxito."
(Charles Chaplin)

Resumo

No cenário de agentes jogadores, é imprescindível que se conte com um apropriado algoritmo de busca apto a tomar decisões acertadas, em tempo satisfatório, em ambientes competitivos nos quais adversários tentam minimizar o êxito dos mesmos. As árvores Alfa-Beta estão na essência da maioria dos atuais algoritmos existentes para tomada de decisão em tais cenários, sendo que a qualidade da solução tende a melhorar na medida em que se eleva o nível de profundidade dessas árvores. Tal fato vem servindo de motivação para um esforço em pesquisa visando a propor versões distribuídas do Alfa-Beta que melhorem o desempenho do algoritmo serial, de forma a permitir a exploração de níveis mais profundos da árvore de busca. Dentre estas versões, destaca-se o APHID (*Asynchronous Parallel Hierarchical Iterative Deepening*), que é um modelo mestre-escravo cujas vantagens principais são a adaptabilidade para operar em arquiteturas de memória distribuída e compartilhada, bem como o fato de prover uma redução das sobrecargas de comunicação e de sincronização em comparação com outras versões distribuídas do Alfa-Beta. Apesar destas contribuições, o APHID apresenta notórias fragilidades no que diz respeito às políticas de: atualização da janela Alfa-Beta dos processadores escravos; prioridade da execução das tarefas a serem executadas por esses mesmos processadores; e *thread* única alocada ao processador mestre para tratar os resultados retornados pelos escravos. Neste contexto, o presente trabalho como metas principais: inspirado no APHID, propor uma nova versão distribuída assíncrona do Alfa-Beta - denominada ADABA (*Asynchronous Distributed Alpha-Beta Algorithm*) - que combata as fragilidades do algoritmo distribuído original; e, objetivando dispor de uma ferramenta para avaliar o desempenho do algoritmo proposto, implementar um sistema multiagente jogador de Damas, denominado D-MA-Draughts, cujo processo de busca seja conduzido por ele. Os resultados dos experimentos, avaliados em arquitetura de memória distribuída, corroboram a melhoria de desempenho provida pelo ADABA, inclusive em termos de taxas de vitória.

Palavras-chave: Algoritmos de Busca. Busca distribuída. Alfa-Beta. APHID. YBWC. ADABA. Jogos Soma Zero. Jogadores Automáticos de Damas. Sistemas Multiagentes.

Abstract

Within the player agent scenario, it is of vital importance that one can count on an appropriate search algorithm, capable of making adequate decisions, in a satisfactory interval, within a competitive setting, in which adversaries try to minimize their outcome. Alpha-Beta trees essentially make up the majority of the current algorithms used today for decision-making in such scenarios, where the quality of the solution tends to improve as the depth elevation of these trees increases. Such a fact has served as the motivation behind research efforts that aim at proposing distributed versions of Alpha-Beta, which improve the performance of the serial algorithm, in such a way as to allow for the exploration of deeper levels of the search tree. Among such versions, the APHID (Asynchronous Parallel Hierarchical Iterative Deepening) is highlighted, which itself is a master-slave model that holds as its main advantages the adaptability to operate in distributed and shared memory architectures, as well as providing a reduction in communication and synchronization overheads, when compared to other distributed versions of Alpha-Beta. In spite of these contributions, APHID presents notorious weaknesses regarding the policies of: updating the Alpha-Beta window of the slave processors; execution priorities of tasks to be executed by these selfsame processors; and *thread* is the only one allocated to the master processor for dealing with results returned by the slaves. In this context, the present study has as its main goals: inspired on APHID, the proposal of a new distributed asynchronous version of the Alpha-Beta – denominated as ADABA (Asynchronous Distributed Alpha-Beta Algorithm) - that combats the weaknesses of the original distributed algorithm; while, making available a tool for the evaluation of the proposed algorithm as its main objective; and implement a multiagent Checkers player system, denominated D-MA-Draughts, whose search process it conducts. The results from the experiments, evaluated in a distributed memory architecture, corroborate the improved performance provided by ADABA, even in terms of victory rates.

Keywords: Search Algorithms. Distributed search. Alpha-Beta. APHID. YBWC. ADABA. Zero-sum games. Automatic Checkers players. Multiagent Systems.

Lista de ilustrações

Lista de tabelas

Lista de siglas

ADABA *Aysnchronous Distributed Alpha-Beta Algorithm*

AIS *ADABA Intermediate Structure*

AM Aprendizagem de Máquina

APHID *Asynchronous Parallel Hierarquical Iterative Deepening*

AR Aprendizagem por Reforço

DI Dinâmica I

DII Dinâmica II

DTS *Dynamic Tree Splitting*

EGA *End Game Agent*

GPU *Graphics Processing Unit*

IA Inteligência Artificial

ID *Iterative Deepening*

IIGA *Initial / Intermediate Game Agent*

LAN *Local Area Network*

MLP *Multi Layer Perceptron*

PV *Principal Variation*

PVS *Principal Variation Splitting*

RNA Rede Neural Artificial

SMA Sistema Multiagente

TD *Temporal Difference*

TT Tabela de Transposição

TTS Tabela de Transposição Sombra

UIDPABS *Unsynchronized Iteratively Deepening Parallel Alfa-Beta Search*

YBWC *Young Brothers Wait Concept*

Sumário

Capítulo 1

Introdução

1.1 Contextualização

Aprendizagem de Máquina (AM) é uma subárea da Inteligência Artificial (IA) amplamente investigada para a construção de sistemas computacionais que, automaticamente, melhoram sua atuação para a resolução de determinados problemas por meio da experiência. Para que esta melhoria seja possível, são desenvolvidas técnicas em que agentes são capazes de extrair conhecimento a partir de amostras de dados ou da observação do ambiente [5].

A Teoria dos Jogos oferece um vasto campo para o estudo das técnicas de AM, visto que se trata de um ramo da matemática aplicada que estuda situações e estratégias em que os jogadores devem escolher diferentes ações no intuito de maximizarem o seu desempenho. Este ramo ganhou atenção especial da área de Ciência da Computação após a publicação do livro *The Theory of Games and Economic Behavior*[6] na segunda metade do século passado. Além disso, nos jogos, é possível encontrar similaridades com problemas encontrados na vida real. Isto ocorre, pois em ambos os casos faz-se necessária a combinação de estratégias para as tomadas de decisão. Além disso, existem situações em que é preciso lidar com obstáculos - representado pelos oponentes, no caso dos jogos - tentando minimizar o efeito das ações realizadas [6]. Assim, é desejável que um agente jogador automático apresente as seguintes características [3]:

- Habilidade para aprender a se comportar em um ambiente onde o conhecimento adquirido é armazenado em uma função de avaliação;
- Capacidade de escolha de um mínimo de atributos possíveis que melhor caracterizem o domínio e que sirvam como um meio pelo qual a função de avaliação adquirirá novos conhecimentos (esta questão é de fundamental importância para se obter agentes com alto desempenho);
- Habilidade para selecionar a melhor ação para um determinado estado ou configura-

ção 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;

- ❏ Ser dotado de poderosas estratégias de aprendizado que facilitem a geração de um agente com bom desempenho.

No caso particular dos jogos do tipo Soma Zero[1], como os jogos de tabuleiro relacionados na Tabela 1, existe a dificuldade adicional de o agente ter de lidar com a complexidade do espaço de estados e do fator de ramificação da árvore de busca (ou árvore de jogo), que impactam fortemente a tomada de decisão. Tal dificuldade está relacionada aos seguintes fatos: 1) nestes jogos, cada configuração do tabuleiro pode ser entendida como um estado de jogo; 2) cada estado pode dar origem a diversos outros a partir da movimentação das peças de um jogador (ramificação de um estado do tabuleiro). Desta forma, estes jogos podem ser solucionados a partir de uma busca recursiva em uma árvore de jogo, contendo um espaço de estados de r^n movimentos, onde r representa o fator de ramificação médio e n a quantidade de níveis (profundidade) do jogo [8]. Neste sentido, a complexidade do espaço de estados é definida pelo número de posições legais (movimentos/estados) que podem ser atingidas a partir da posição inicial do jogo. O fator de ramificação é a média de estados que podem originar a partir de uma posição do tabuleiro [9].

Tabela 1 – Complexidade de alguns jogos de tabuleiro em relação ao fator de ramificação e espaço de estados[4]

Jogo	**Fator de Ramificação**	**Espaço de Estados**
Xadrez	30 - 40	10^{50}
Dramas	8 - 10	10^{17}
Gamão	$\pm$420	10^{20}
Othelo	$\pm$5	$< 10^{30}$
Go 19x19	$\pm$360	10^{160}
Abalone	$\pm$80	$< 3^{61}$

Estes fatos reforçam os argumentos que tornam os jogos de tabuleiro um domínio extremamente rico a ser explorado em AM. Nesta direção, diversos trabalhos reportados na literatura implementaram e/ou propuseram técnicas de AM tanto no contexto de aprendizagem supervisionada (quando o processo de aprendizagem é fortemente norteado por especialistas humanos ou por bases de dados), quanto no contexto da não-supervisionada (quando o agente aprende baseado essencialmente em suas próprias experiências) [10].

[1] Jogos de Soma Zero (*Zero-Sum Games*) na Teoria dos Jogos são aqueles em que o ganho de um jogador representa a perda de outro. Pode haver mais de um jogador em um jogo de Soma Zero, no entanto, apenas um pode atingir o objetivo da vitória. Nas situações em que não existe um vencedor, há um empate [7].

Dentre tais trabalhos, pode-se citar [11, 12, 13, 14] no Xadrez, [15, 16, 17, 18, 19, 20, 21, 22] no jogo de Damas, [23] no Othelo e, [8, 24, 25, 26] no domínio do jogo do Go.

Dentre as técnicas de AM empregadas em problemas como os dos jogos da Tabela 1, o algoritmo de busca Alfa-Beta pode ser destacado. Ele permite ao agente prever jogadas boas e ruins em uma árvore de busca do jogo. Este algoritmo é uma otimização do clássico Minimax, pois elimina da árvore de jogo inúmeras subárvores desnecessárias que não podem conter a solução - processo denominado como poda alfa-beta [27]. Salienta-se que a aplicação deste algoritmo não é restrita a apenas o domínio dos jogos, ele pode ser empregado em qualquer problema em que o ganho de um equivale à perda de outro - o que caracteriza um cenário de Jogo de Soma Zero [6]. No entanto, é no domínio dos jogos que este algoritmo é fortemente estudado e aplicado.

O Alfa-Beta retorna a solução ótima nas situações em que é possível explorar toda a árvore de busca. Todavia, em problemas com elevado espaço de estados, esta exploração é inviável. Neste sentido, buscando melhorar a acurácia da solução provida pelo Alfa-Beta permitindo explorar mais estados no espaço do problema, alguns recursos foram propostos para serem agregados a este algoritmo. Dentre eles cita-se a busca por aprofundamento iterativo - do inglês, *Iterative Deepening* (ID) - e as Tabela de Transposição (TT) [28]. Na busca por ID são realizadas diversas passagens (varreduras) pela árvore de busca em um intervalo de tempo ou limitação de profundidade, de modo que a cada passagem a profundidade da busca é incrementada. TTs são repositórios que armazenam estados já avaliados a fim de evitar reprocessamento. O uso de ID e TT em conjunto permite a utilização de heurísticas para ordenação da árvore de busca, o que pode aumentar o número de podas nas situações em que o melhor movimento está localizado mais à esquerda [29].

1.2 Motivação

Nos jogos existe uma forte relação entre a visão em profundidade (*look-ahead*) que é possível atingir na árvore de jogo e a qualidade dos movimentos do jogador. Desta forma, a fim de permitir que o algoritmo busque em profundidades cada vez maiores, permitindo uma maior exploração do espaço de estados, utiliza-se a distribuição do algoritmo Alfa-Beta aliado aos recursos de TT e ID. Esta distribuição permite a execução do algoritmo de modo paralelo. Todavia, a tarefa de paralelizar o Alfa-Beta não é trivial, uma vez que ele tem uma natureza serial[2]. Por esta razão, a versão paralela deve lidar com algumas fontes de ineficiência, tais como [1]: sobrecarga de sincronização, que ocorre quando um

[2] No Alfa-Beta, as informações mais precisas para a execução das podas advêm dos ramos mais à esquerda da árvore de busca que formam os limites da busca - denominados de *alfa* (mínimo) e *beta*(máximo) - em que a solução ótima estará contida. Desta forma, para que exista a maximização das podas, é necessário que a exploração de um ramo apenas inicie após a exploração dos ramos a sua esquerda.

processador necessita aguardar por informações provenientes de outro processador; sobrecarga de comunicação que está associada com a frequência da troca de mensagens entre os processadores; e sobrecarga de busca, que ocorre quando são explorados nós desnecessariamente, uma vez que não há informações disponíveis que permitiriam a ocorrência de podas.

Estas fontes de ineficiência não são independentes, por exemplo, aumentando a sobrecarga de comunicação poderia ajudar a reduzir a sobrecarga de busca [1]. Todavia, um equilíbrio entre estas fontes de ineficiência é difícil de encontrar. Assim, as propostas de distribuição do Alfa-Beta buscam um *trade-off* para maximizar sua eficácia. Nesse sentido, diversos algoritmos tem sido propostos na literatura baseados em duas abordagens de paralelismo - síncrona e assíncrona - operando em arquiteturas de memória compartilhada ou distribuída [1, 12, 13, 30, 31, 32, 33]. Dentre eles, destacam-se o síncrono *Young Brothers Wait Concept* (YBWC) [33] e o assíncrono *Asynchronous Parallel Hierarquical Iterative Deepening* (APHID)[1].

Cada abordagem de distribuição apresenta pontos fortes e limitações, por exemplo, as abordagens síncronas apresentam a característica de tentar criar um fluxo para que o algoritmo execute o mais próximo possível da versão serial. Para isso, diversos pontos de sincronização de informações são estabelecidos a fim de identificar alterações nos limites da janela de busca alfa-beta. No caso particular do YBWC, quando alguma alteração dos limites da janela é identificada, uma mensagem é enviada para todos os processadores para que eles possam atualizar suas janelas de busca locais com os novos limites visando aumentar o número de podas. Apesar deste procedimento ser bastante pertinente, o sincronismo pode fazer com que processadores fiquem ociosos em alguns momentos. Além disso, manter todos os processadores atualizados aumenta substancialmente o volume de mensagens trocadas, o que acarreta em sobrecarga de comunicação.

Em contrapartida, as abordagens assíncronas apresentam como vantagem em relação às síncronas os seguintes fatos: tentam manter todos os processadores ocupados durante todo o tempo da busca; e visam diminuir o volume de trocas de mensagens entre os processadores, o que reduz a sobrecarga de comunicação. Contudo, no que se refere à redução de troca de mensagens deve-se considerar que efeitos indesejáveis são passíveis de ocorrer. Por exemplo, no APHID uma das estratégias adotadas para que ele consiga esta redução é o emprego de uma política de atualização da janela de busca Alfa-Beta lastreada em uma constante cujo valor é estipulado empírica e manualmente. No caso, os limites *alfa* e *beta* serão calculados a partir desse valor. Dessa forma, se tal valor produz limites excessivamente estreitos para a janela de busca, o algoritmo pode perder a solução ótima que seria apontada pelo Alfa-Beta serial. No entanto, se o valor da constante produzir limites exageradamente amplos para a janela de busca, o algoritmo pode apresentar sobrecarga de busca por deixar de detectar inúmeras possibilidades de poda.

As abordagens distribuídas do Alfa-Beta podem operar em arquiteturas de memória compartilhada ou distribuída. De fato, as de memória compartilhada são mais apropriadas para a execução deste tipo de algoritmo, uma vez que permitem acesso de todos os processadores envolvidos às informações globais - por exemplo, da TT - armazenadas na memória. No entanto, arquiteturas de memória compartilhada possuem o custo mais expressivo do *hardware*, que aumenta proporcionalmente em relação ao número de núcleos de processamento. Logo, uma arquitetura de memória compartilhada adequada a um ambiente de alto poder de processamento é menos acessível economicamente. Por outro lado, arquiteturas de memória distribuída apresentam menor custo, pois podem ser obtidas a partir da conexão de um conjunto de microcomputadores conectados por meio de uma rede local (*Local Area Network* (LAN)). O YBWC produz melhor desempenho em arquitetura de memória compartilhada [34], ao passo que o APHID foi concebido de modo a produzir bons resultados tanto em arquitetura de memória compartilhada quanto distribuída [1].

Considerando os argumentos apresentados, esta tese propõe um novo algoritmo de busca distribuído baseado no Alfa-Beta, o *Aysnchronous Distributed Alpha-Beta Algorithm* (ADABA). Este algoritmo, conforme o nome sugere, implementa a abordagem de distribuição assíncrona e segue o modelo de paralelismo mestre-escravo. A dinâmica de distribuição de tarefas adotada foi baseada nas ideias do APHID [1]. De fato, a incitação maior do ADABA é aprimorar os seguintes pontos do APHID que podem comprometer a performance da busca:

- ❏ A política de formação da janela de busca, conforme dito anteriormente, é dependente de uma constante definida empírica e manualmente;

- ❏ No curso de uma busca a partir de um determinado nó n, o mestre realiza diversas varreduras por uma subárvore s que tem como raiz n e é expandida até um nível d_m (que é inferior à profundidade máxima d da busca). A cada varredura, são utilizadas as informações recebidas dos escravos referentes aos nós presentes no nível d_m. Desta forma, é atribuição do mestre gerenciar as soluções retornadas pelos escravos. No APHID, esta tarefa é realizada por uma única *thread* vinculada ao processo mestre. Tal situação pode resultar no reprocessamento de s sem que estejam disponíveis todas as soluções produzidas pelos escravos até aquele instante. Assim, pode haver o comprometimento da velocidade do processo de busca, e consequentemente, da profundidade máxima que poderia ser alcançada no caso do uso da estratégia por ID afetando a qualidade da solução final provida pelo algoritmo.

- ❏ Quando o mestre atinge a profundidade d_m da busca, ele distribui os nós presentes neste nível entre os escravos. Cada escravo deve priorizar as tarefas que vai explorar de acordo com alguma política. No APHID, a maior prioridade é dada aos nós que foram explorados em profundidades mais rasas. Caso todos os nós possuam a

mesma profundidade associada, é considerada a localização do nó na árvore de busca onde são priorizados os ramos presentes mais a esquerda. Considerando uma árvore ordenada, os ramos mais a esquerda possuem mais proximidade com a solução da busca. Sendo assim, a política do APHID é primordialmente baseada na quantidade de trabalho a ser realizado ao invés da "distância da solução", o que pode acarretar em sobrecarga de busca, pois as soluções dos nós mais "promissores" podem demorar mais a serem obtidas.

O ADABA visa atacar tais pontos de fragilidade do APHID por meio das seguintes contribuições:

- Automatiza o processo de atualização das informações da janela-busca nos processadores escravos. Neste caso, diferentemente do método empírico e manual adotado no APHID, o ADABA define uma nova política para formar e atualizar a janela de busca entre os processadores escravos. Para isso, ele explora conceitos inerentes à versão síncrona YBWC, em que há a definição de uma janela de busca inicial a partir da exploração do nó localizado mais a esquerda de uma árvore de busca ordenada. Desta forma, o ADABA utiliza as informações providas pelo ramo mais a esquerda da árvore de busca do mestre para obter parâmetros confiáveis para o cálculo dos limites da janela de busca que será utilizada pelos processadores escravos.

- Cria um grupo de *threads* inerentes ao processo mestre para gerenciar as tarefas recebidas dos escravos. Desta forma, o mestre passa a ter mais informações acessíveis referentes à avaliação dos nós presentes no nível d_m a cada varredura em sua subárvore.

- Implementa uma política para o processo de priorização das tarefas que serão executadas pelos processadores escravos focada essencialmente na "distância da solução" dos nós considerando que eles estão ordenados no modo "melhor-para-pior", isto é, aqueles mais a esquerda representam melhores soluções do que aqueles a direita na árvore de jogo.

Um ponto a se destacar é que, devido à forma como as abordagens assíncronas de distribuição operam, a solução da busca pode ser diferente daquela ótima que seria retornada pela versão serial do algoritmo nas mesmas circunstâncias. No entanto, como as versões distribuídas operam em níveis mais profundos da árvore de busca, a qualidade da solução pode ser compensada. Por isso, as contribuições do ADABA visam buscar um equilíbrio entre eficiência do tempo de resposta do algoritmo e a qualidade da solução mantendo uma boa taxa de correspondência com a solução ótima do Alfa-Beta (aquela que a versão serial retornaria).

O ADABA é proposto como uma biblioteca que pode operar tanto em arquiteturas de memória compartilhada, quanto distribuídas. Além disso, ele é independente do problema

de aplicação podendo ser utilizado em outros domínios com características semelhantes ao de um jogo do tipo Soma Zero.

Neste trabalho, o problema de aplicação escolhido para testar a performance do ADABA foi o jogo de Damas. Tal escolha deve-se aos seguintes fatos:

1. o jogo de Damas, inegavelmente, caracteriza um excelente laboratório de estudo de técnicas de AM devido à sua elevada complexidade do espaço de estados, conforme pode ser observado na Tabela 1;

2. a autora do presente trabalho desenvolveu em trabalhos anteriores duas arquiteturas de sistemas jogadores de Damas automáticos, são elas:

    a) o D-VisionDraughts [19], trata-se de um sistema monoagente que consiste de uma Rede Neural Artificial Multicamadas que aprende por reforço por meio do método das Diferenças Temporais TD(λ) aliada à estratégia de treino por *self-play* com clonagem. Para tomadas de decisão ele utiliza o algoritmo YBWC;

    b) o D-MA-Draughts [3], trata-se de um Sistema Multiagente (SMA) composto por 26 agentes com arquitetura análoga à do D-VisionDraughts. Nele há 1 agente que atua na partida durante as fases de início e intermédio (IIGA - *Initial Intermediate Game Agent*), isto é, enquanto o tabuleiro possui mais de 12 peças. Os demais agentes (25), denominados como de final de jogo, atuam na partida apenas no estágio final (quando o tabuleiro tem, pelo menos, 12 peças). Esta versão do D-MA-Draughts conta com duas dinâmicas de atuação em partidas no que diz respeito à iteração entre os agentes nas fases finais do jogo: a primeira dinâmica envolve menor cooperação entre os agentes, ao passo que a segunda há maior cooperação. Além disso, convém ressaltar que a arquitetura disponível para a execução do D-MA-Draughts foi do tipo memória distribuída.

Neste sentido, este trabalho apresenta a proposta adicional de melhorar a arquitetura do SMA D-MA-Draughts por meio de duas vertentes: 1) estabelecer a melhor dinâmica de cooperação entre os agentes nas fases finais do jogo que melhorem o seu desempenho em disputas; 2) inserir o ADABA como motor de tomadas de decisão para os agentes do SMA a fim de melhorar a arquitetura individual de seus agentes.

Convém salientar que, recentemente, em Ref. [14], os autores utilizaram como motor de tomada de decisão o algoritmo de busca em árvores Monte-Carlo (MCTS - *Monte-Carlo Tree Search*) ao invés do Alfa-Beta no jogo de Xadrez - que apresenta as mesmas particularidades do jogo de Damas, a saber: regras dependentes da posição das peças no tabuleiro; aumento das possibilidades de movimento (ramificações) com o andamento da partida; o jogo pode finalizar como vitória, derrota ou empate. Até então, o Alfa-Beta

era o algoritmo de busca mais adequado para lidar com este tipo de problema. Os resultados reportados em Ref. [14] demonstram o excelente nível de jogo de um jogador de Xadrez utilizando MCTS. No entanto, o sucesso desse jogador não está restrito à utilização do MCTS, está no fato dele estar aliado a outras técnicas, tais como as redes neurais convolutivas [35], o que resultou no algoritmo de aprendizagem não supervisionada *AlphaZero*. Este último corresponde a um algoritmo que deriva das técnicas de aprendizagem empregadas no jogador de Go *AlphaGo Zero* [26]: primeiro agente automático com aprendizagem sem supervisão humana a derrotar um campeão mundial neste jogo. De fato, toda a proposta de [14, 26], indiscutivelmente, trouxeram excelentes avançados para o campo da AM e poderão auxiliar na proposta e condução de trabalhos futuros a esta tese de doutorado.

1.3 Objetivos e Desafios da Pesquisa

Considerando os pontos apresentados na seção 1.2, este trabalho tem como objetivo geral:

1. propor um novo algoritmo de busca distribuído assíncrono para o Alfa-Beta, o *Aysnchronous Distributed Alpha-Beta Algorithm* (ADABA), que estabeleça um equilíbrio na sobrecarga de busca permitindo obter uma boa taxa na qualidade da solução retornada quando comparado à solução ótima (aquela que seria provida pela versão serial em uma mesma situação) e que seja portável tanto para arquiteturas de memória compartilhada quanto distribuída. Para isso, esta versão visa incorporar características positivas das abordagens síncrona, a partir da exploração do algoritmo YBWC, e assíncrona, por meio da exploração do algoritmo APHID.

2. produzir uma nova versão do SMA jogador de Damas D-MA-Draughts a partir da melhoria da arquitetura individual de seus agentes em que o processo de tomada de decisão seja conduzido pelo ADABA. Além disso, deseja-se que tal SMA opere com a dinâmica de jogo mais adequada a produzir bons resultados em partidas. Nesta direção, o D-MA-Draughts é utilizado como instrumento para avaliar a eficácia do algoritmo proposto.

1.3.1 Objetivos Específicos

A fim de alcançar os objetivos gerais deste trabalho de doutorado, os seguintes objetivos específicos devem ser obtidos:

1. Identificar pontos positivos e negativos das abordagens dos algoritmos síncrono YBWC e assíncrono APHID a fim de nortear a proposta do novo algoritmo foco deste trabalho.

2. Propor a nova versão distribuída do algoritmo Alfa-Beta, o ADABA, de modo que tal versão tire proveito dos pontos positivos identificados no objetivo específico 1.

3. Construir o sistema monoagente jogador de Damas ADABA-Draughts a fim de validar o desempenho do algoritmo proposto em um problema de alta complexidade. Ressalta-se que a arquitetura monoagente tratada neste objetivo corresponde à arquitetura individual dos agentes do SMA D-MA-Draughts.

4. Determinar a dinâmica de jogo mais adequada a produzir bons resultados em uma partida que deve ser adotada pelo D-MA-Draughts. Neste caso, será considerada a sua primeira versão [3], uma vez que o cumprimento deste objetivo específico deve ser realizado em paralelo aos estudos referentes ao objetivo específico 1.

5. Inserir a arquitetura do jogador ADABA-Draughts na arquitetura multiagente do D-MA-Draughts, obtendo assim uma nova versão do SMA jogador de Damas considerando a dinâmica definida no objetivo específico 4.

1.4 Hipótese

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

1. Na IA, o processo de busca por boas ações para solucionar um problema é demasiadamente complexo quando a dimensão do espaço de estados é elevada e o agente enfrenta obstáculos minimizando seu sucesso. Nessas situações, o algoritmo Alfa-Beta combinado com TT e ID é bastante utilizado. Assim sendo, é possível melhorar o desempenho do Alfa-Beta aumentando o *look-ahead* da exploração da árvore de busca através de uma nova versão distribuída? Considerando que a tarefa de distribuir o Alfa-Beta não é fácil devido à necessidade de compartilhar as informações da janela de busca, é possível que esta nova versão busque maximizar a utilização dos recursos de hardware disponíveis através da abordagem assíncrona de paralelismo? Esta nova versão terá capacidade de atuar na otimização do balanceamento das fontes de ineficiência que podem acometer uma versão distribuída do Alfa-Beta, como, por exemplo, sobrecarga de comunicação e de busca, sem perder o foco na qualidade da solução (dada pela taxa de correspondência com a versão serial), uma vez que não basta um algoritmo rápido, se a solução retornada é de baixa acurácia?

2. Para a construção de sistemas distribuídos existem dois tipos de arquiteturas de computadores disponíveis: memória compartilhada e memória distribuída. Devido ao custo expressivo relacionado aos equipamentos de arquitetura de memória compartilhada, a aquisição deles é frequentemente inviabilizada. Deste modo, é possível

a construção de um algoritmo que se adeque bem a ambos os tipos de arquitetura fazendo com que a limitação de *hardware* não seja fator de impedimento para usufruir dos benefícios da distribuição do algoritmo Alfa-Beta?

3. O SMA D-MA-Draughts é um jogador que possui uma versão preliminar que utiliza a versão distribuída síncrona do Alfa-Beta YBWC como motor para suas tomadas de decisão. É possível melhorar a performance do jogador ao atualizar seu motor de busca com um algoritmo que proporcione melhor acurácia nas tomadas de decisão em um ambiente de arquitetura de memória distribuída?

4. O jogador D-MA-Draughts atua em uma partida segundo duas dinâmicas. Cada uma delas envolve um meio de cooperação, mais constante ou não, entre os agentes. Existe entre tais dinâmicas aquela mais adequada a produzir bons resultados em partidas?

1.5 Contribuições Científicas

As principais contribuições do presente trabalho são:

- Criação de uma nova versão distribuída assíncrona do algoritmo Alpha-Beta, denominada ADABA, que é inspirada no APHID e que conta com estratégias para reduzir as fragilidades do algoritmo APHID;

- Criação de uma biblioteca de busca composta pelo algoritmo Alfa-Beta versão *Fail-Soft* e pela versão distribuída proposta neste trabalho, o ADABA. Tal biblioteca permite o emprego destes algoritmos em outros sistemas que utilizem o Alfa-Beta para resolução de problemas. Desta forma, existe a opção de escolha entre a versão serial e distribuída.

- Criação do sistema monoagente jogador de Damas automático APHID-Draughts cujo mecanismo de busca conta com a abordagem assíncrona de paralelismo do Alfa-Beta (APHID).

- Criação do sistema monoagente jogador de Damas automático ADABA-Draughts cujo mecanismo de busca conta com a abordagem assíncrona de paralelismo do Alfa-Beta proposta neste trabalho (ADABA).

- Obtenção de uma plataforma multiagente, o D-MA-Draughts, que estende a arquitetura do ADABA-Draughts a todos os agentes e utiliza a melhor dinâmica de cooperação entre os agentes em partidas.

É importante destacar que o algoritmo produzido no contexto deste trabalho, o ADABA, pode ser estendido a outros problemas que sejam caracterizados pela perspectiva de que

as ações de um agente venham a minimizar os efeitos das ações de um outro agente, assim como ocorre nos jogos do tipo Soma Zero. Para isso, é necessário a implementação dos métodos de geração dos estados sucessores da árvore de busca, bem como da função de avaliação. Tais requisitos são necessários, pois são casos particulares do problema tratado.

1.6 Organização da Tese

Esta tese está organizada conforme disposto a seguir:

Capítulo 2 relaciona todo o referencial teórico referente à construção do algoritmo de busca distribuído proposto, bem como à arquitetura dos agentes jogadores de Damas contemplados neste trabalho. Desta forma, este Capítulo abrange os seguintes tópicos: Sistema Multiagente (SMA); Rede Neural Artificial (RNA); Aprendizagem por Reforço e o Método das Diferenças Temporais; Técnica de Treinamento por Self-play com Clonagem; Estratégias de Busca; Algoritmos de Busca Seriais; Busca Paralela; e a Problemática do Jogo de Damas que relaciona a representação do estados de tabuleiro e a questão da transposição de estados em Damas.

Capítulo 3 apresenta o estado da arte dos trabalhos correlatos. Para isso, primeiramente é apresentado os principais algoritmos de busca distribuídos baseados no Alfa-Beta. Na sequência, apresenta-se os principais jogadores automáticos de Damas, os quais estão divididos de acordo com o modo de aprendizado adotado: supervisionado e não-supervisionado.

Capítulo 4 realiza uma comparação conceitual entre as abordagens de distribuição síncrona e assíncrona do algoritmo de busca Alfa-Beta representadas, respectivamente, pelas versões YBWC e APHID.

Capítulo 5 apresenta o algoritmo proposto neste trabalho, o ADABA, destacando seu funcionamento e contribuições.

Capítulo 6 traz a arquitetura geral do monoagente ADABA-Draughts, que foi utilizada para avaliar o desempenho dos algoritmos baseados no Alfa-Beta tratados neste trabalho.

Capítulo 7 apresenta o SMA D-MA-Draughts enfatizando as suas dinâmicas de jogo, a sua arquitetura multiagente e a expansão do ADABA-Draughts a todos os agentes do sistema.

Capítulo 8 apresenta todos experimentos e análise dos resultados realizados no decorrer deste trabalho de doutorado.

Capítulo 9 encerra esta tese apresentando as conclusões, contribuições científicas, produções bibliográficas, limitações encontradas e sugestões de trabalhos futuros.

Capítulo 2

Referencial Teórico

Neste capítulo é apresentado o referencial teórico para guiar na compreensão das técnicas e estratégias utilizadas no cumprimento dos objetivos deste trabalho de doutorado.

2.1 Agentes e Sistemas Multiagentes

Segundo Russel e Norving [10], um agente é uma entidade que percebe o ambiente em que está inserido e age sobre este ambiente a fim de atingir algum objetivo. Um agente é definido como qualquer entidade que pode perceber um ambiente através de sensores e agir, de maneira autônoma, por meio de atuadores.

Um Sistema Multiagente (SMA) é um sistema computacional em que dois ou mais agentes interagem entre si de forma a desempenhar determinadas tarefas e alcançarem, em conjunto, um determinado objetivo. O comportamento global do SMA deriva da interação entre os agentes que o compõem. Nesse sentido, deve-se buscar por uma sincronia que permita a esses agentes a capacidade de coordenar seus conhecimentos, objetivos e habilidades de uma forma conjunta, em favor da execução de uma ação ou da resolução de algum problema, o qual depende da cooperação entre os agentes.

Nos sistemas em que há cooperação entre os agentes, cada um pode atuar em uma determinada parte do problema e interagir com os demais para tentar atingir um objetivo ou auxiliar os demais a alcançarem seus objetivos. Nestes casos, os agentes apresentam duas características fundamentais, a saber: são capazes de agir de forma autônoma tomando decisões que os levam à satisfação de seus objetivos; e podem interagir com outros agentes no intuito de coordenar, cooperar, competir e/ou negociar as ações que os levam à satisfação do objetivo [36].

Particularmente, o emprego de SMAs em ambientes de cooperação tem atraído a atenção de muitos pesquisadores em uma variedade de temas. Por exemplo, em Ref. [37] os autores fizeram uma revisão de diversos trabalhos relacionados ao problema de coordenação de múltiplos veículos não tripulados. O sistema CILIOS, apresentado em [38], é um SMA em que a arquitetura é baseada em redes neurais simbólicas para lidar

com o problema de realização de monitoramento mútuo em SMAs autônomos. Em Ref. [39] e Ref. [40] são apresentadas abordagens baseadas em técnicas evolucionárias para melhorar a efetividade de sistemas de recomendação. Os autores de [41] apresentam um SMA colaborativo que utiliza busca Monte Carlo para otimizar o consumo de eletricidade. Neste trabalho, o SMA proposto constitui-se de um sistema colaborativo em que os agentes coordenam suas ações em fases finais do jogo de Damas.

Um SMA colaborativo pode estar inserido em um ambiente de competição. Neste caso, dentre os diversos agentes que atuam no ambiente, existem aqueles que estão relacionados cooperando em prol de um objetivo, e outros agentes que visam o mesmo objetivo, todavia, apenas uma das partes terá êxito. No caso do jogo de Damas, por exemplo, o ambiente do jogo é composto por um agente e um oponente, todavia, um destes agentes pode ser representado por um SMA onde há diversos agentes trabalhando em conjunto (cooperando) para maximizar seu desempenho na partida de modo a atingir seu objetivo de vitória.

Moulin e Chaib-Draa [42] evidenciam as características que constituem vantagens significativas dos SMA sobre um sistema de único agente, entre elas: maior eficiência na resolução de problemas; mais flexibilidade de ação por possuir agentes de diferentes habilidades agrupados para resolver problemas; e aumento da segurança pela possibilidade de agentes assumirem responsabilidades de agentes que falham.

2.2 Estratégias de Busca

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

De forma genérica, as estratégias de busca tradicionais envolvem uma busca em uma árvore de jogo que descreve todos os estados possíveis a partir de um estado inicial dado. Formalmente, o espaço de busca é constituído por um conjunto de nós conectados por arcos. Cada arco pode ou não estar associado a um valor que corresponde a um custo c de transição de um nó a outro. A cada nó é associado uma profundidade p, sendo que a mesma tem valor 0 (zero) no nó raiz e aumenta de uma unidade para um nó filho. A aridade a de um nó é a quantidade de filhos que o mesmo possui, e a aridade de uma árvore é definida como a maior aridade de qualquer um de seus nós. O objetivo da busca é encontrar um caminho (ótimo ou não) do estado inicial até um estado final explorando, sucessivamente, os nós conectados aos nós já explorados até a obtenção de uma solução para o problema. Neste contexto, a estratégia de busca pode seguir diversas abordagens sobre como os nós da árvore serão explorados. Dentre elas é possível citar a busca em

profundidade limitada e a busca em aprofundamento iterativo.

2.2.1 Busca em Profundidade Limitada

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

Na busca em profundidade limitada, também conhecida como busca de profundidade primeira, é inserido um limite máximo que pode ser atingido na exploração de um nó da árvore. Portanto, mesmo que o nó explorado ainda tenha sucessores a serem expandidos, se o limite for antigido, a busca não prossegue e retrocede a fim de iniciar a busca em outro nó [10].

2.2.2 Busca com Aprofundamento Iterativo

A qualidade de um programa jogador que utiliza um algoritmo de busca, como o Alfa-Beta (veja seção 2.3.2), depende muito do número de jogadas que ele pode olhar adiante (*look-ahead*). Para jogos com um fator de ramificação grande (como os da Tabela 1), o jogador pode levar muito tempo para pesquisar poucos níveis adiante.

A maioria dos sistemas jogadores utilizam mecanismos para delimitar o tempo máximo permitido de busca. Todavia, se for utilizado um algoritmo de busca em profundidade, não existe garantia de que a busca irá se completar antes do tempo máximo estabelecido. Para evitar que o tempo se esgote e o programa jogador não possua nenhuma informação de qual a melhor jogada a ser executada, busca em profundidade não deve ser utilizada [43].

Larry Atkin [44] introduziu a técnica de aprofundamento iterativo (do inglês *Iterative Deepening* (ID)) como um mecanismo de controle do tempo de execução durante a expansão da árvore de busca. É possível notar que a ideia básica do aprofundamento iterativo é realizar uma série de buscas, em profundidade, independentes, cada uma com um *look-ahead* acrescido de um nível. Assim, é garantido que o procedimento de busca iterativo encontre o caminho mais curto para a solução justamente como a busca em largura encontraria [10]. Caso o tempo se esgote e o algoritmo ainda não tenha encontrado uma solução "ótima", a busca será interrompida e o último estado analisado será retornado.

2.3 Algoritmos de Busca Seriais

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

2.3.1 Algoritmo Minimax

O Minimax [10] é uma técnica de busca para determinar a estratégia ótima em um cenário de jogo com dois jogadores. O objetivo dessa estratégia é decidir a melhor jogada para um dado estado do jogo. Há dois jogadores no Minimax: o MAX e o MIN. Uma busca em profundidade é feita a partir de uma árvore onde a raiz é a posição corrente do jogo. As folhas dessa árvore são avaliadas pela ótica do jogador MAX, e os valores dos nós internos são atribuídos de baixo para cima com essas avaliações. As folhas do nível minimizar são preenchidas com o menor valor de todos os seus filhos, e o nível maximizar são preenchidos com o maior valor de todos os nós filhos.

A Figura 1 mostra um exemplo de aplicação do algoritmo Minimax que gera a árvore de busca do jogo para um determinado estado.

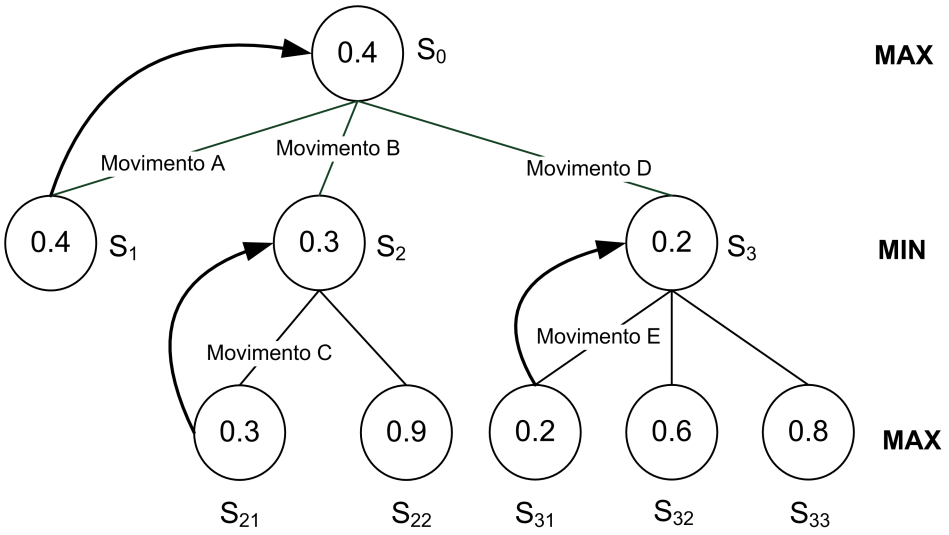


Figura 1 – Árvore de busca expandida pelo algoritmo Minimax

Conforme é possível notar, os valores das folhas 0.3 e 0.2 são retornadas para os nós S_2 e S_3, respectivamente, uma vez que estes nós estão em um nível de minimização, ou seja, recebem o menor valor de seus filhos. O valor 0.4 é retornado para S_0 que é a raiz da árvore e está em um nível de maximização (recebe o maior valor de seus filhos). Desta forma, na Figura 1, o melhor movimento que o agente deve escolher é o *movimento A*.

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

2.3.2 Algoritmo Alfa-Beta

O algoritmo Alfa-Beta otimiza o algoritmo Minimax por eliminar seções da árvore que não podem conter a melhor predição [27], [45]. Por exemplo, na Figura 2, o algoritmo Alfa-Beta, diferentemente do algoritmo Minimax (exemplificado na Figura 1) detecta que não há necessidade de avaliar as predições dos nós destacados. Portanto, o melhor movimento (*movimento A*) é encontrado mais rapidamente.

Este algoritmo recebe tal nome devido a dois valores: alfa e beta. Estes valores delimitam o intervalo que o valor da predição do melhor movimento correspondente à entrada do estado de tabuleiro deve pertencer. Desta forma, para os sucessores de um nó n, as seguintes situações podem ocorrer [10]:

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

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

O algoritmo Alfa-Beta possui uma natureza intrinsecamente serial, uma vez que depende do conhecimento prévio para evitar a busca em partes da árvore que não têm influência no resultado final. Ele conta com duas versões chamadas *hard-soft* e *fail-soft* que diferem apenas no valor da predição associada aos nós (ou estados de tabuleiro) dos subproblemas relacionados à raiz da árvore. Neste contexto, o valor correspondente à raíz da árvore (retorno final do algoritmo) é o mesmo valor minimax para ambas as versões [46], [47]. As subseções a seguir apresentam estas duas variantes do algoritmo Alfa-Beta.

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

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

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

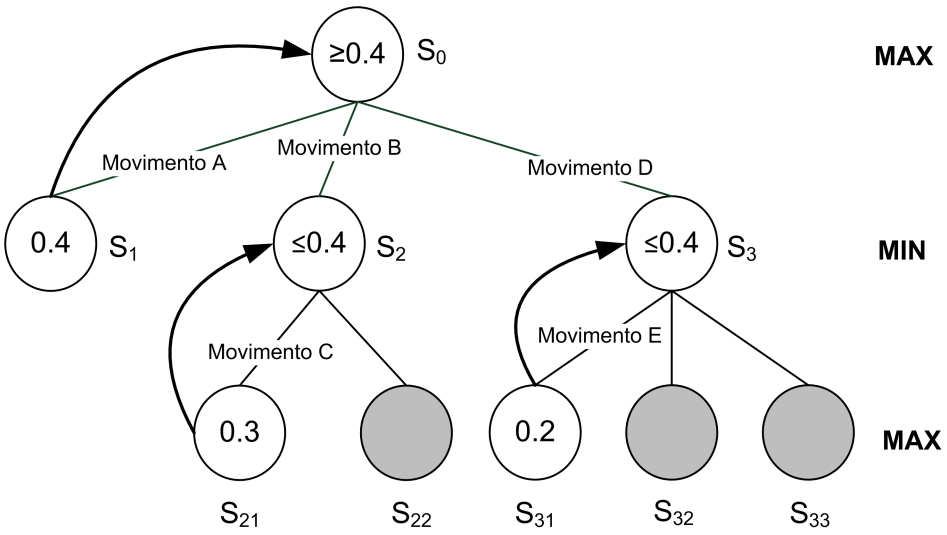


Figura 2 – Árvore de busca expandida pelo algoritmo Alfa-Beta na versão hard-soft

Portanto, o valor retornado representa ou o valor minimax ou o limite imposto pelo intervalo alfa-beta. Por exemplo, na Figura 2, o valor ≤ 0.4 será retornado para o nó S_2 e S_3 indicando que com os movimentos B ou D, o valor da predição de ambos os nós irá ser **pelo menos** 0.4. De fato, após calcular que os valores dos sucessores mais à esquerda dos nós S_2 e S_3 tiveram predições 0.3 e 0.2, respectivamente, o algoritmo detecta que não é necessário explorar os nós destacados na Figura 2 (que causam uma poda). Por esta razão, se eles tiverem uma predição inferior ao de seu irmão mais à esquerda, suas predições seriam candidatas a "subirem" para seus pais (que são nós minimizadores). No entanto, de qualquer forma, eles não poderiam ser escolhidos pelo nó maximizador S_0 que já tem disponível o valor 0.4 produzido pelo nó S_1. Por outro lado, se eles tiverem uma predição superior a seu irmão mais a esquerda, eles não podem ser escolhidos por seus pais (nós minimizadores).

A variante *hard-soft* possui uma limitação quando deseja-se melhorar a performance do Alfa-Beta através da utilização de Tabela de Transposição (TT). Uma TT é um repositório onde são armazenadas predições associadas a nós já avaliados que podem ser utilizados quando um nó aparecer na busca outra vez. Por exemplo, considerando novamente a

Figura 2, se a versão *hard-soft* fosse utilizada, seria armazenado a predição 0.4 para os nós S_2 e S_3 na TT. Certamente, este valor 0.4 existente na TT poderia não ser o valor real da predição destes nós, mas dos limites impostos pelo intervalo alfa-beta. Portanto, se a variante *hard-soft* for utilizada em conjunto com uma TT para calcular o melhor movimento, e durante este processo, os estados S_2 e S_3 ocorrerem novamente, as suas respectivas predições seriam buscadas na TT. Todavia, caso estes estados encontrados na TT pudessem ser utilizados (ao cumprirem um conjunto de restrições detalhadas na seção 5.8.1) poderia ser realizado um movimento diferente do escolhido pelo algoritmo Minimax.

Em algoritmos de busca que pretendem utilizar TTs como meio de aprimoramento é utilizada a variante *fail-soft* do Alfa-Beta, que será detalhada na seção 2.3.2.2 a seguir.

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

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

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

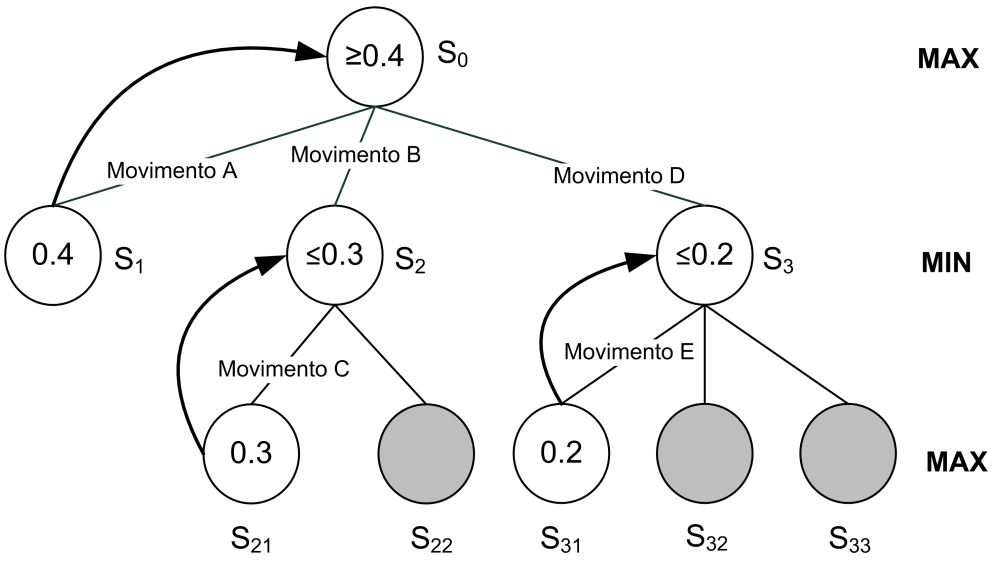


Figura 3 – Árvore de busca expandida pelo algoritmo Alfa-Beta na versão fail-soft

A versão *fail-soft* é muito parecida com a versão *hard-soft*, diferindo apenas no valor das predições associadas aos nós dos subproblemas procedentes da raiz da árvore, fato que permite sua integração com TT. A versão *fail-soft* retorna o verdadeiro valor minimax

para cada um dos subproblemas existentes na expansão da árvore de busca, ou seja, o valor calculado sempre representa um limite do valor minimax. Por exemplo, na Figura 3, quando o algoritmo observa o movimento C, ele retorna o valor 0.3 (apesar do fato de 0.3 estar fora do intervalo alfa-beta), o que significa que o movimento B possui um valor de predição igual a 0.3 [17]. Portanto, se esta versão do algoritmo Alfa-Beta for utilizada em conjunto com TT, caso um estado S_i esteja na tabela e este valor satisfaça um conjunto de restrições (detalhadas na seção 5.8.1) o valor retornado é correspondente ao valor minimax. Desta forma, o resultado final do algoritmo não sofre alterações, ou seja, possui o mesmo retorno caso a árvore fosse expandida fazendo uso do algoritmo Minimax.

2.3.2.3 Otimizando o Alfa-Beta a partir da Ordenação da Árvore de Busca

A fim de maximizar o número de podas que podem ser efetuadas a partir do algoritmo Alfa-Beta é empregada a ordenação da árvore de jogo. Uma árvore é dita perfeitamente ordenada se os ramos estiverem ordenados da esquerda para a direita. Os nós de uma árvore perfeitamente ordenada podem ser classificados em três tipos [48]:

Nós tipo PV: são também denominados nós do tipo 1. É o ramo mais a esquerda da árvore de busca e é denominado de variação principal (no inglês *Principal Variation* (PV)). Todos os nós PV são buscados com a janela de busca $(-\infty, +\infty)$. Uma vez que os limites da janela são infinitos, podas nunca ocorrem em nós do tipo PV e todos os seus ramos são explorados. O primeiro sucessor de um nó PV também é do tipo PV, os demais são do tipo CUT.

Nós tipo CUT: também são conhecidos como nós do tipo 2. São sucessores dos nós do tipo PV (exceto o primeiro) e ALL. Uma vez que um nó do tipo CUT não é o primeiro sucessor de um nó PV, ele irá ter os limites estabelecidos pelo nó do primeiro ramo do tipo PV. Considerando um nó PV em um nível de maximização, o segundo ramo a ser expandido neste nó é um tipo CUT de minimização. A avaliação retornada pelo primeiro nó PV serve como limite inferior para o nó CUT. Neste caso o nó CUT não tem limite superior, uma vez que não houve determinação no nó PV de maximização - um nó PV determina apenas um dos dois limites. Uma vez que a árvore pesquisada é perfeitamente ordenada, a primeira ramificação procurada em um nó CUT conduz imediatamente a uma poda. O primeiro sucessor de um nó CUT é do tipo ALL, os demais são do tipo CUT.

Nós tipo ALL: também denominados como nós do tipo 3. São sucessores de nós CUT. Como é o primeiro ramo de um nó CUT, ele contém as mesmas informações de janela de seu pai. Considerando um exemplo de um nó CUT minimizador em que existe um limite inferior de janela válido e que o limite superior é infinito, o primeiro ramo

a ser expandido produz um nó ALL maximizador. Uma vez que o limite superior é infinito no nó ALL, nenhuma poda pode ser feita e todos os ramos são pesquisados. Devido a árvore perfeitamente ordenada, as avaliações retornadas por nós sucessores do tipo ALL não são suficientes para estreitar o limite inferior da janela de busca - as avaliações retornadas são piores do que as estabelecidas pelos nós PV. De fato, a ordenação em nós do tipo ALL é irrelevante. Os sucessores de um nó ALL são do tipo CUT.

A Figura 4 apresenta um exemplo de uma árvore perfeitamente ordenada destacando o tipo de cada um dos nós.

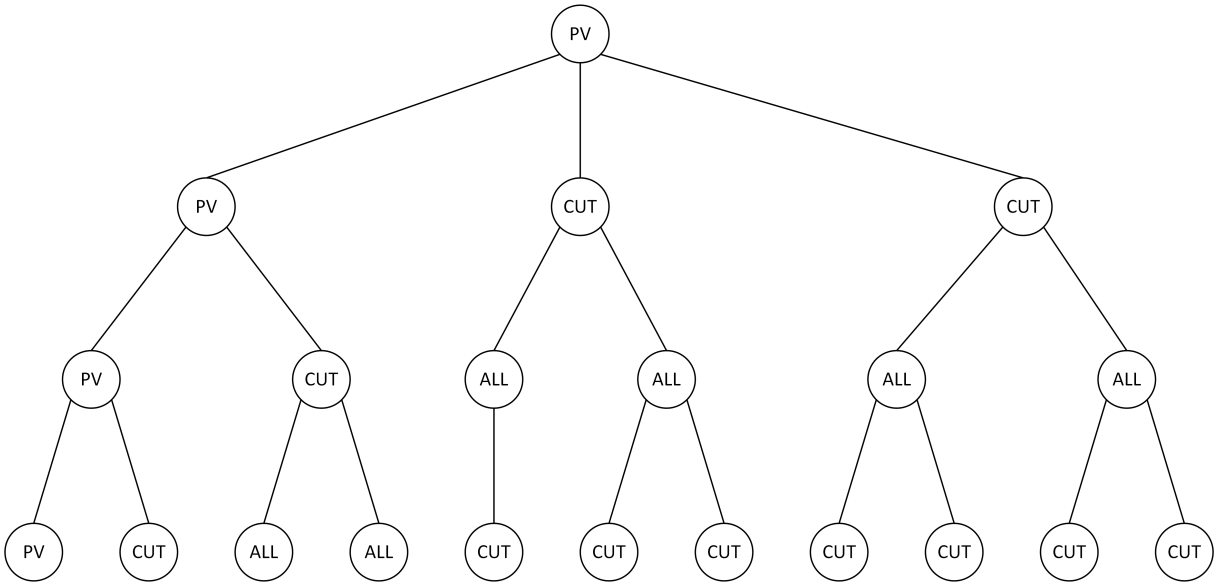


Figura 4 – Tipos de nós em uma árvore de jogo [1].

Uma árvore perfeitamente ordenada cuja a janela de busca inicia em $(-\infty, +\infty)$ é denominada de *árvore mínima*, uma vez que apenas os nós necessários para completar a busca são explorados [27].

A obtenção de uma árvore perfeitamente ordenada em problemas de jogos é muito difícil, uma vez que a predição de um nó apenas é conhecida quando ele é alcançado. Por esta razão, neste tipo de problema a ordenação é parcial e utiliza alguns mecanismos auxiliares, tais como a utilização da estratégia de busca por ID (seção 2.2.2).

2.4 Busca Paralela

Um algoritmo paralelo pode ser executado por diferentes processadores em um mesmo espaço de tempo, retornando a mesma saída que seria gerada caso tal algoritmo fosse executado sequencialmente. O objetivo desses algoritmos é melhorar o desempenho das

versões sequenciais, visto que, normalmente, são empregados em problemas que exigem alta carga de processamento.

A paralelização de um algoritmo pode ocorrer de acordo com duas abordagens: síncrona e assíncrona. A primeira estabelece diversos pontos de centralização (sincronização) de dados das tarefas distribuídas (mesmo que não estejam concluídas) cujo propósito é obter informações que podem ser úteis para o andamento global do algoritmo. Por outro lado, nas abordagens assíncronas, as tarefas são processadas autonomamente e, apenas após concluídas existe um retorno com a solução que é enviada para o ponto de origem (de distribuição das tarefas). A definição de qual abordagem utilizar depende intrinsecamente da natureza do problema.

O uso de algoritmos paralelos de tomada de decisão em jogos permite realizar buscas mais profundas, fato que propicia uma melhor visão futura de jogo (*look-ahead*) ao agente no momento de escolher um movimento. Jogadores com um campo de visão maior têm mais condições de fazer melhores escolhas e, consequentemente, de jogarem em um nível superior a jogadores com um campo de visão do espaço de estados menor.

Como regra, quanto mais profunda for a busca, melhor é a qualidade da jogada, pois o erro introduzido pela função de avaliação é reduzido. Para se obter um nível de profundidade extra no jogo de Damas, por exemplo, geralmente é requerido o aumento da velocidade da busca por um fator de duas vezes (no xadrez este fator varia entre quatro e oito vezes) [48], [49]. Neste contexto, as subseções a seguir são dedicadas a apresentar conceitos referentes à busca paralela em árvores de jogo, que é o foco do presente trabalho. Alguns exemplos deste tipo algoritmos de busca (com enfoque na busca Alfa-Beta paralela) serão apresentados na seção 3.1.

2.4.1 Medidas de desempenho

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

$$S = T_s/T_p \tag{1}$$

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

O *speedup* não é normalizado para o número de processadores utilizados. Dessa forma, há um conceito derivado do *speedup* denominado eficiência. A eficiência representa quanto do potencial de *speedup* foi efetivamente alcançado, ou o quão bem o sistema foi utilizado.

Em sistemas paralelos ideais, o *speedup* é menor que o número de processadores e a eficiência é igual a um. Na prática, o *speedup* é menor que o número de processadores e a eficiência fica entre zero e um [49]. De um modo geral, é possível dizer que a eficiência mede a qualidade do algoritmo paralelo. A eficiência E executada em n processadores é definida como:

$$E = S/n \tag{2}$$

2.4.2 Dificuldades Enfrentadas na Busca Paralela

A paralelização de árvores de jogo é uma tarefa difícil [48], visto que, dentre os algoritmos desenvolvidos até o momento não há um que apresente uma eficiência ótima (ou seja, próxima ou igual a 1). Assim, é importante entender alguns problemas que cercam este tipo de busca resultando em certa ineficiência na busca paralela. Neste contexto, é possível destacar três fontes de ineficiência [1], [49]:

Sobrecarga de busca: refere-se aos nós explorados pelo algoritmo paralelo que não seriam explorados pelo algoritmo serial. Esta sobrecarga (*overhead*) ocorre primariamente pela carência de informação dos processadores. Em algoritmos do tipo *branch-and-bound* como o Alfa-Beta, a informação obtida na busca de um ramo da árvore pode causar uma poda em outros ramos. Assim, se os sucessores de um nó são expandidos paralelamente, um nó pode não tirar vantagem de toda informação que estaria disponível na busca serial, resultando em busca desperdiçada.

Sobrecarga de sincronização: ocorre quando um processador é forçado a esperar por informações provindas de outros processadores.

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

2.4.3 Tabela de Transposição em ambiente Distribuído

Em um algoritmo paralelo a implementação de uma TT pode ocorrer de duas formas:

1. *Compartilhada:* todos os processadores tem permissão de leitura e de escrita na tabela, pois compartilham a mesma memória RAM. Neste caso, a TT também pode ser chamada de TT global.

2. *Distribuída:* cada processador tem a sua própria TT, visto que os processadores não compartilham da mesma memória RAM. Neste caso, para manter as informações

sincronizadas em todos os processadores seria necessário uma troca de informações elevada, fato que resultaria em uma sobrecarga de comunicação muito grande, o que é inviável para garantir a boa performance do algoritmo. Este tipo de TT também é denominada de TT local. A desvantagem deste tipo de TT é que pode haver reprocessamento de estados, uma vez que um estado e_i que já foi processado em um processador P_i, pode aparecer em outro processador P_j. Todavia, como P_i e P_j não tem acesso aos mesmos dados na memória, P_j teria que reprocessar e_i.

A Figura 5 ilustra a comunicação de acesso entre os processadores em uma arquitetura que possui uma TT global (Figura 5(a)), bem como a comunicação entre os processadores que possuem TT distribuídas (Figura 5(b)). Conforme é possível observar, quando há uma TT global, se um processador P_1 explorou um estado e_1 e, em outro momento o mesmo estado e_1 apareceu para o processador P_n, este último pode recuperar os dados da exploração de e_1 evitando seu reprocessamento. Por outro lado, no mesmo caso, em uma infraestrutura de TTs distribuídas, e_1 seria reprocessado, uma vez que P_n não teria acesso aos estados armazenados na TT de P_1. Logo, a utilização de uma TT global é a solução ideal para problemas como o do jogo de Damas, visto que todos os processadores tem acesso aos estados já explorados por outros processadores. Contudo, este tipo de TT requer uma infraestrutura de memória compartilhada que, em geral, está disponível com equipamentos de hardware com custo financeiro muito elevado. Neste cenário, em situações onde não se dispõe de tais equipamentos, resta a alternativa de uso das TTs distribuídas.

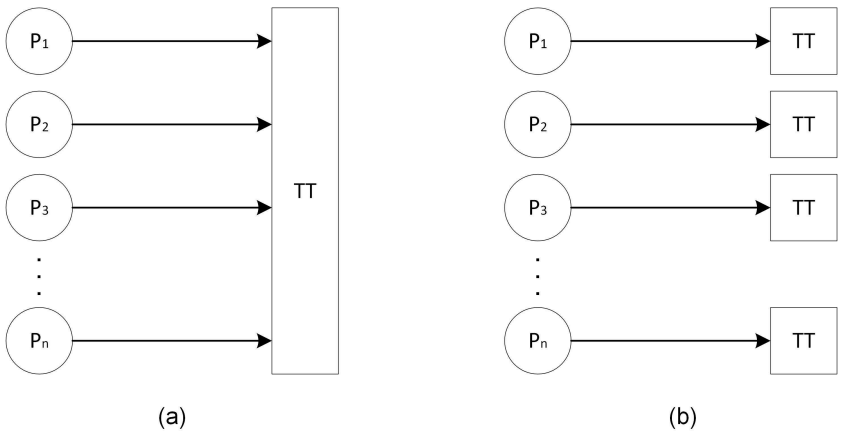


Figura 5 – (a) Comunicação dos processadores com uma TT global; (b) Comunicação dos processadores com TTs distribuídas.

2.5 Problemática do Jogo de Damas

O jogo de Damas (*Checkers* no inglês americano e *Draughts* no inglês britânico), é um tradicional jogo de tabuleiro de estratégia disputado entre dois jogadores. O tabuleiro

é formado por quadrados em que cada um constitui uma posição. A configuração mais comum é de 8×8 posições - existem variações como o de 10x10 posições. Os quadrados são dispostos por uma cor clara e uma escura alternadamente. Apenas os quadrados de cor escura são utilizados no jogo. Cada jogador possui 12 peças que são diferenciadas por cores, em que as mais comuns são preto, vermelho ou branco. As peças de cada jogador são posicionadas em lados opostos. A Figura 6 apresenta a configuração inicial padrão para um jogo de Damas 8x8 em que um dos jogadores atuará com peças pretas e o outro com vermelhas. O objetivo do jogo é que um jogador consiga capturar todas as peças de seu oponente, ou pelo menos, o deixe sem opções de movimento.

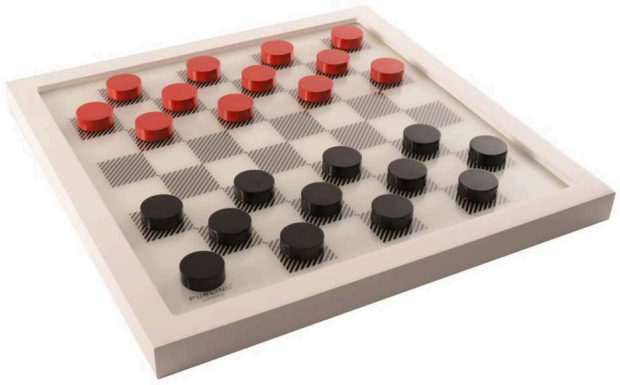


Figura 6 – Configuração inicial padrão para um jogo de Damas 8×8

A fim de apresentar mais detalhes concernentes à problemática do jogo de Damas, a seção 2.5.1 apresenta as regras deste jogo, a seção 2.5.2 mostra duas formas para representar um tabuleiro para Damas - vetorial e matricial -, por fim, a seção 2.5.3 aborda a questão da transposição de estados em Damas.

2.5.1 Regras do Jogo de Damas

Existem variações das regras do jogo de Damas em alguns países, todavia, as regras mais comuns são as inglesas, inclusive, são as utilizadas neste trabalho. Assim sendo, as regras relativas ao jogo de Damas inglesas são [50]:

- O tabuleiro deve ser posicionado de tal forma que cada jogador tenha um quadrado claro no canto direito mais próximo dele, conforme indicado na Figura 7.
- As peças devem estar posicionadas nos quadrados escuros;
- O jogador preto sempre inicia a partida;
- Os movimentos ocorrem na diagonal, um quadrado por vez, a menos que o movimento envolva uma captura.

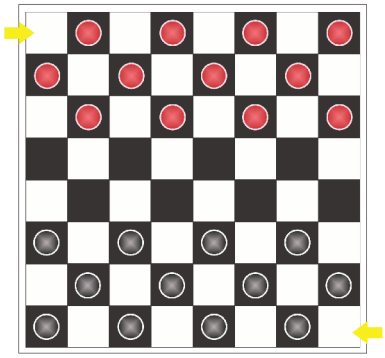


Figura 7 – Disposição inicial das peças no tabuleiro de Damas

- Peças simples podem avançar apenas a frente no sentido de seu oponente, conforme ilustrado na Figura 8.

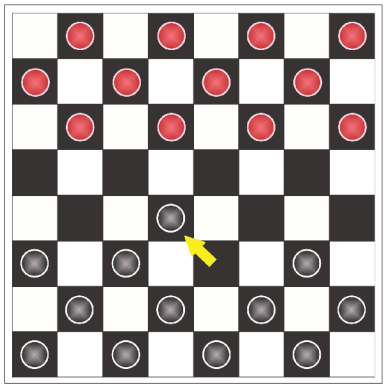


Figura 8 – Representação de um movimento de uma peça preta

- Se uma peça do oponente estiver próxima na diagonal de uma peça do jogador corrente ocorre uma situação de captura. Neste caso, o jogador corrente deve saltar a peça do oponente que será removida do tabuleiro. A Figura 9 apresenta esta situação em que uma peça preta irá capturar uma peça vermelha.

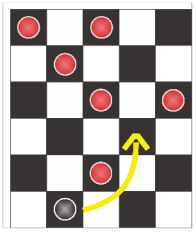


Figura 9 – Representação de uma captura

- Uma captura pode ser múltipla, isto é, logo após o salto, o jogador corrente verifica que é possível realizar outra captura a partir da mesma peça. A Figura 10 ilustra essa situação. Todas as capturas possíveis devem ser realizadas.

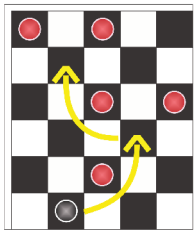


Figura 10 – Representação de uma captura múltipla

- ❏ Quando existir possibilidade de captura, o jogador deve efetua-la.
- ❏ Se houver mais de uma possibilidade, então o jogador pode escolher a que deseja realizar.
- ❏ Quando uma peça atinge a linha da borda oposta de sua posição de jogada, ela é promovida a rainha. Para esta representação, uma peça capturada (de mesma cor) é colocada sobre ela, de modo que ela fique duas vezes mais alta que uma peça simples.
- ❏ Rainhas também podem mover-se apenas um quadro diagonalmente, todavia, elas podem realizar o movimento para frente ou para trás.
- ❏ As capturas realizadas por uma rainha podem ser feitas em qualquer direção, inclusive para trás. Desta forma, em uma captura múltipla podem ocorrer situações cujo movimento seja feito para frente e, na sequência, para trás, aumentando as possibilidades de capturas.
- ❏ O jogo finaliza quando o oponente não tem mais posições de movimento a realizar, seja porque todas suas peças foram capturadas, ou porque suas peças ficaram bloqueadas pelas peças do oponente.

2.5.2 Representação do Tabuleiro nos Jogadores de Damas

Existem diversas maneiras de representar internamente um tabuleiro de jogo de Damas [51]. Particularmente, a representação interna de um tabuleiro neste trabalho é feita de duas maneiras: representação vetorial e representação por características, também chamada de NET-FEATUREMAP.

2.5.2.1 Representação Vetorial

A escolha de uma estrutura de dados para representar o tabuleiro em um jogo é fundamental para a eficiência de um jogador automático de Damas. Algoritmos de busca em profundidade, baseados no Alfa-Beta, estão presentes nos melhores projetos da história dos jogos de tabuleiro; e a escolha adequada de uma estrutura de dados para representar

o tabuleiro afeta, consideravelmente, a velocidade de execução desse tipo de algoritmo. A representação vetorial é recomendada para casos em que um algoritmo Alfa-Beta é usado em virtude de sua precisão na representação de um estado de tabuleiro.

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

Pseudocódigo 1 Representação dos possíveis valores das posições de um tabuleiro em Damas.

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

Pseudocódigo 2 Representação do tabuleiro para o jogo de Damas 8x8

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

O tabuleiro é uma estrutura do tipo BOARD implementada como um vetor de 32 elementos do tipo BOARDVALUES. Cada elemento do vetor BOARD representa uma casa do tabuleiro e cada casa do tabuleiro possui um dos seguintes valores presentes em BOARDVALUES:

EMPTY: posição com ausência de peças;

BLACKMAN: posição que contem uma peça preta;

REDMAN: posição que contem uma peça vermelha;

BLACKKING: posição que contem uma peça do tipo rainha preta;

REDKING: posição que contem uma peça do tipo rainha vermelha.

2.5.2.2 Representação NET-FEATUREMAP

A representação NET-FEATUREMAP representa um determinado estado de tabuleiro utilizando um conjunto de características que capturam conhecimentos relevantes

a cerca do domínio do jogo de Damas. A utilização de características para este tipo de representação foi inicialmente proposta por Samuel [52], com o intuito de prover medidas numéricas para melhor representar as diversas propriedades de posições de peças sobre um tabuleiro. Várias dessas características implementadas por Samuel resultaram de análises feitas sobre o comportamento de especialistas humanos em partidas de Damas. Em termos práticos, essas análises tinham o objetivo de tentar descobrir quais características referentes a um estado do tabuleiro são frequentemente analisadas e selecionadas pelos próprios especialistas quando vão escolher seus movimentos de peças durante uma partida de Damas. Neste sentido, as características fornecem medidas quantitativas e qualitativas para melhorar a representação de diversas propriedades posicionais das peças sobre o tabuleiro.

A presente proposta utiliza um subconjunto de 14 dessas características, as quais estão descritas na Tabela 2. Cada característica é representada por um valor absoluto correspondente à quantidade de peças que a representa no tabuleiro. Este valor absoluto é convertido em bits os quais, em conjunto com os bits das outras características, constituem a saída do mapeamento NET-FEATUREMAP.

A conversão de um determinado tabuleiro vetorial T_i para NET-FEATUREMAP é definido pelo mapeamento C:

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

$C(T_i) = < f_1(T_i),..., f_n(T_i) >$, e

$f_j(T_i) = a_j$, em que:

n é a quantidade de características f_j $(1 \leq j \leq n)$ usadas na representação do tabuleiro por características, a representa a presença (se $a \geq 0$) e quantidade de peças que representam a característica f_i no tabuleiro T_i. Isso significa que cada tabuleiro T_i é representado por uma *n-upla* composta de n atributos, os quais correspondem as características $f_i, ..., f_n$, respectivamente.

Por exemplo, considera-se f_1 como sendo a *feature* peças em vantagem "*PieceAdvantage*", que verifica em T_i a quantidade de peças em vantagem de um jogador. Se o jogador para o qual a *feature* está sendo verificada possui 5 peças em vantagem em relação ao seu oponente, o atributo $f_1(T_i)$ produz 5 (ou seja, $a = 5$); caso ele não possua peças em vantagem, o valor do atributo $f_1(T_i)$ é 0 (ou seja, $a = 0$). Se um tabuleiro possui peças em vantagem para um jogador, consequentemente, ele não possui peças em desvantagem para esse mesmo jogador. A Figura 11 ilustra como seria a representação do tabuleiro em relação às características *PieceAdvantage* e *PieceDisadvantage*, ambas ocupando a 1ª e a 2ª posição do tabuleiro, respectivamente, considerando 5 peças em vantagem.

O presente trabalho adotou este mapeamento para representar os estados de tabuleiro na entrada da RNA, conforme será apresentado na seção 6.3.

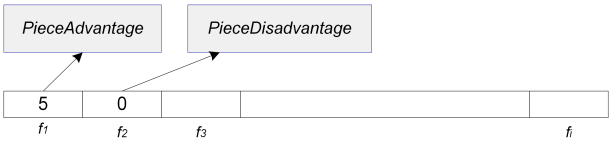


Figura 11 – Exemplo de tabuleiro representado por NET-FEATUREMAP

CARACTERÍSTICAS	DESCRIÇÃO FUNCIONAL	BITS
PieceAdvantage	Contagem de peças em vantagem para o jogador preto (no caso, número de peças que o jogador tem a mais).	4
PieceDisavantage	Contagem de peças em desvantagem para o jogador preto (no caso, número de peças que o jogador tem a menos).	4
PieceThreat	Total de peças pretas que estão sob ameaça.	3
PieceTake	Total de peças vermelhas que estão sob ameaça de peças pretas.	3
Backrowbridge	Se existe peças pretas nos quadrados 1 e 3 e se não existem rainhas vermelhas no tabuleiro.	1
CentreControl	Total de peças pretas no centro do tabuleiro.	3
XCentreControl	Total de quadrados no centro do tabuleiro onde tem peças vermelhas ou para onde elas possam mover.	3
TotalMobility	Total de quadrados vazios para onde as peças vermelhas podem mover.	4
Exposure	Total de peças pretas que são rodeadas por quadrados vazios em diagonal.	3
Advancement	Total de peças pretas que estão na $5^{\underline{a}}$ e $6^{\underline{a}}$ linha do tabuleiro menos as peças que estão na $3^{\underline{a}}$ e $4^{\underline{a}}$ linha.	4
DoubleDiagonal	Total de peças pretas que estão na diagonal dupla do tabuleiro.	3
KingCentreControl	Total de rainhas pretas no centro do tabuleiro.	3
DiagonalMoment	Total de peças vermelhas que estão localizadas na posição 1 ou na posição 2 de uma diagonal dupla mais as peças passivas que estão na ponta da diagonal.	3

Threat	Total de posições para qual uma peça preta pode se mover e assim poder ameaçar uma peça vermelha em um movimento subsequente.	3

Tabela 2 – Conjunto de características (*features*) utilizadas neste trabalho.

2.5.3 Ocorrência de Transposição em Damas

Em um jogo de Damas, pode-se chegar a um mesmo estado do tabuleiro várias vezes e, quando isso ocorre, diz-se que houve uma transposição [45]. As transposições ocorrem, em Damas, de duas maneiras básicas:

- *Diferentes combinações de jogadas com peças simples*: as peças simples não se movem para trás, apesar disso, elas podem desencadear uma transposição, como mostrado na Figura 12. Nesse caso, os estados do tabuleiro mostrados em a e d são idênticos, assim como os estados mostrados em c e f. Assumindo a como estado inicial, é possível alcançar c passando por b. Assumindo d como estado inicial, é possível alcançar f passando por e. Então, os únicos estados diferentes são b e e. No caso da sequência de movimentos a, b e c, o jogador preto move-se primeiro para a direita e, em seguida, para a esquerda, enquanto na sequência de movimentos d, e e f, o jogador preto move-se primeiro para a esquerda e, em seguida, para a direita.

- *Diferentes combinações de jogadas com rainhas*: as rainhas se movem em qualquer direção, gerando transposições com facilidade, conforme mostrado na Figura 13. Partindo do estado a, avançando a rainha, é possível alcançar o estado b e, em seguida, recuando a rainha, é possível alcançar o estado c, idêntico ao a.

É importante ressaltar que os agentes para Damas podem ter seu desempenho melhorado se contarem com um repositório para armazenar estados do tabuleiro que já tenham sido avaliados de modo a utilizar aqueles que reapareçam por transposição. Dessa forma, eles não precisam ser reavaliados. Este tipo de repositório é denominado de Tabela de Transposição (TT).

2.6 Redes Neurais Artificiais

Uma Rede Neural Artificial (RNA) é um modelo computacional, baseado em redes neurais biológicas, que consiste em uma rede de unidades básicas simples chamadas neurônios (nós). Estes nós são neurônios artificiais, originalmente criados para simular a operação de uma célula do cérebro [35].

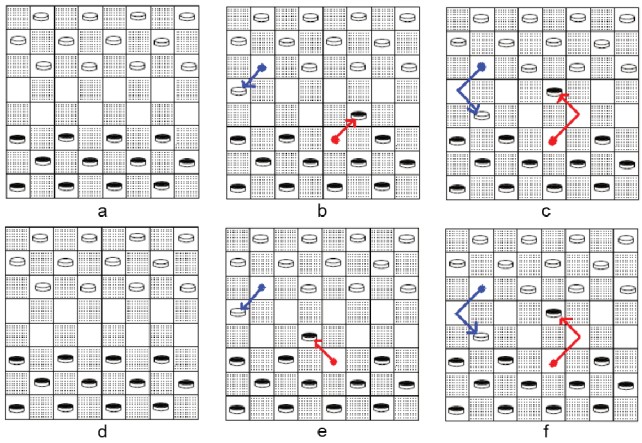


Figura 12 – Exemplo de transposição em c e f: o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com peças simples

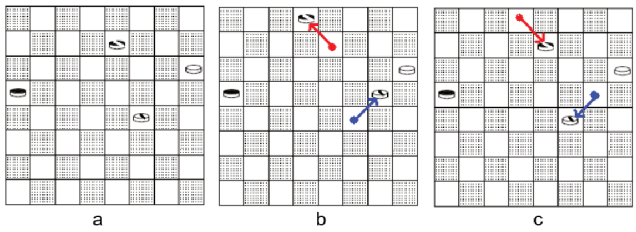


Figura 13 – Exemplo de transposição em a e c: o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com rainhas

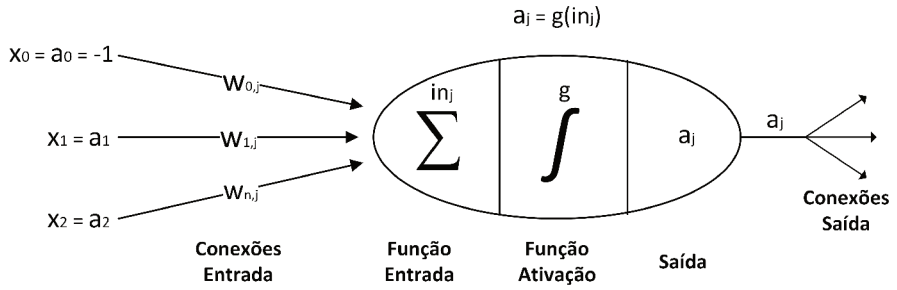


Figura 14 – Célula Neural Artificial

McCulloch e Pitts [53] propuseram em 1943 o primeiro modelo matemático para um neurônio artificial. Basicamente, este modelo, ilustrado na Figura 14, é composto por 3 elementos:

1. um conjunto de *elos de conexão:* conexões dos sinais de entrada, representados por $x_1, x_2, ..., x_n$, aos pesos da rede, representados por $w_{1j}, w_{2j}, ..., w_{nj}$. Estes pesos representam a força de conexão entre a unidade de processamento n que gera a entrada x_n e o neurônio j;

2. um *combinador linear* responsável por combinar os elos de conexão através da função

in_j, formando o campo local induzido.

3. uma *função de ativação* g que tem por meta restringir a amplitude de saída do neurônio.

Esse modelo ainda possui um *threshold* ou *bias* que corresponde a um valor arbitrário representado pelo sinal de entrada x_0 que é conectado ao neurônio j com peso w_{0j}. Este 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 restrição (3) é satisfeita:

$$\sum_{i=1}^{n} w_{i,j} \cdot x_i > w_{0,j} \cdot x_0. \tag{3}$$

Matematicamente é possível calcular a saída a_j correspondente ao neurônio j por meio da equação 4:

$$a_j = g(in_j) = g(\sum_{i=0}^{n} w_{i,j} \cdot x_i). \tag{4}$$

É importante ressaltar que a função de ativação g deve ser projetada de acordo com as necessidades do problema. Existem diversas funções de ativação na literatura apropriadas para cada tipo de problema: funções *threshold*, *ramp*, *signum*, *hyperbolic tangent* e outras [35].

Uma RNA é construída de tal forma a ser capaz de solucionar um problema. Para isso, ela deve ser treinada, isto é, ela deve ser capaz de aprender como tratar o problema para o qual foi projetada. O processo de treinamento consiste no ajuste dos pesos sinápticos até encontrar conexões adequadas entre seus neurônios. O objetivo é produzir sinais de saída desejáveis relacionados aos sinais de entrada que são apresentados para a RNA. Portanto, o ajuste dos pesos sinápticos entre os neurônios de uma RNA representa o aprendizado de cada neurônio. Isto significa que cada neurônio, em conjunto com todos os outros, representam a informação que passa pela rede. Desta forma, o conhecimento associado aos neurônios, e consequentemente, à própria rede, reside nos pesos sinápticos [35].

2.6.1 Redes Neurais Multicamadas

Os Perceptrons Multi-Camadas (*Multi Layer Perceptron* (MLP) - do inglês *Multilayer Perceptron*) são RNAs que se caracterizam pela presença de uma camada de entrada, conectada a uma ou mais camadas intermediárias (ou camadas ocultas), e uma camada de saída. Nas camadas ocultas, os neurônios são apenas unidades de processamento. Assim, neste modelo, o sinal de entrada se propaga para frente através da rede camada por camada, isto é, a saída do conjunto de neurônios de cada camada da rede é usado

como entrada para a camada seguinte. Esta é uma generalização de uma RNA de uma única camada (perceptron linear), onde existe uma modificação do padrão linear em que uma ou mais camadas com funções de ativação não lineares [10] são utilizadas. Com a adição de uma ou mais camadas intermediárias, o poder computacional não linear e de memória da rede são aumentados[35].

A Figura 15 mostra uma representação de uma MLP. Cada saída a_j^m relativa ao j-ésimo neurônio de saída da camada m é dada na equação 5:

$$O_j = g(in_j) = g(\sum_{i=0}^{n} w_{i,j}^{m-1} \cdot a_i^{m-1}) = a_j^m, \tag{5}$$

onde a_i^{m-1} é a saída do i-ésimo neurônio da camada $m-1$ conectada ao neurônio de saída j com peso $w_{i,j}^{m-1}$. O parâmetro in_j é o campo local induzido da entrada do j-ésimo neurônio. Isto significa que o valor O_j (saída da rede MLP) depende dos pesos da rede. O objetivo de um treinamento supervisionado de uma RNA é ajustar os pesos de tal forma a minimizar a diferença (chamada *erro*) entre os valores O_j obtidos e a saída desejada. Por esta razão, tal diferença é usada como um parâmetro no cálculo do ajuste dos pesos da RNA [35]. Por outro lado, no processo de aprendizado não supervisionado, onde o valor desejado não está disponível, esta diferença é substituída pela diferença entre as duas saídas sucessivas produzidas pela RNA conforme processo detalhado na seção 6.3.1.

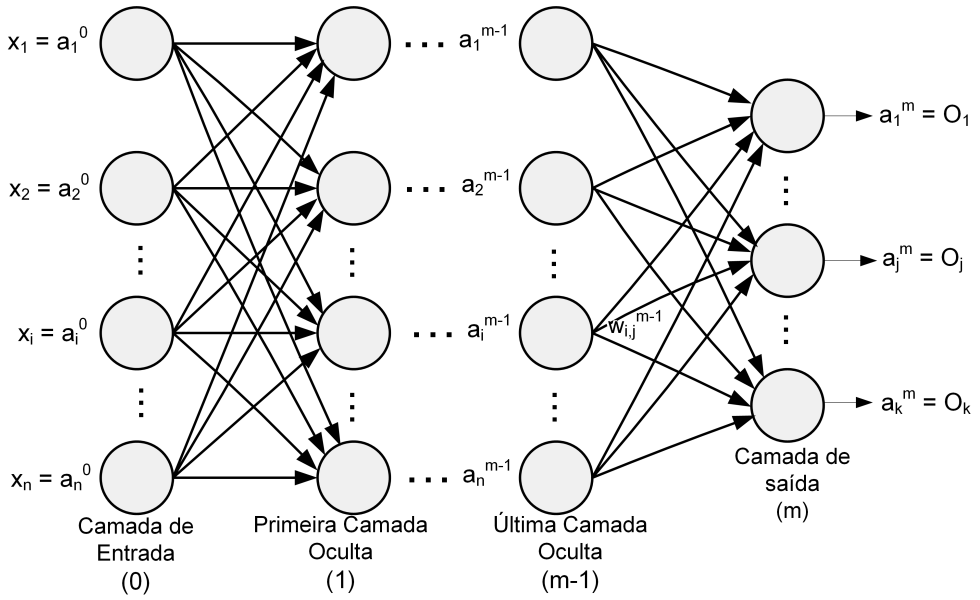


Figura 15 – Perceptron Multicamadas

2.6.2 Redes Neurais Auto Organizáveis - Kohonen-SOM

As RNAs Kohonen-SOM (*Kohonen Self-Organizing Maps*), desenvolvidas por Teuvo Kohonen [54] [55], pertencem a uma classe de RNA denominada de Redes Neurais Competitivas. A arquitetura destas RNAs, apresentada na Figura 16, é composta por duas

camadas: uma camada de entrada - representada pelos neurônios X_n da Figura 16; e uma camada de saída ou competitiva - representada pelos neurônios Y_m da Figura 16. Todos os neurônios entre estas camadas são interconectados através dos pesos sinápticos (w_{nm} na Figura 16). Os neurônios da camada competitiva competem entre si a representação do dado apresentado à camada de entrada. A competição é feita a partir do cálculo da menor distância (Distância Euclidiana) entre os neurônios da camada de entrada e cada neurônio da camada de saída. Ao final da competição, o neurônio vencedor e seus vizinhos têm seus pesos atualizados. O fator de atualização dos pesos da vizinhança são definidos conforme o problema e é conhecido como função de vizinhança.

O aprendizado das redes Kohonen-SOM ocorre de modo não supervisionado. Após a etapa de aprendizado, onde os pesos são ajustados, existe uma fase de utilização em que não há alterações nos pesos.

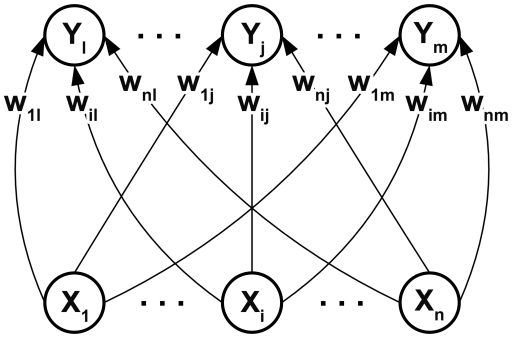


Figura 16 – Arquitetura de uma rede Kohonen-SOM

2.7 Aprendizagem por Reforço e o Método das Diferenças Temporais

Dentro do paradigma da Aprendizagem de Máquina (AM), as abordagens tradicionais que utilizam aprendizagem supervisionada trabalham com sistemas que aprendem através de exemplos de pares de entrada e saída. Tais pares fornecem aos sistemas indicativos de como se comportar para tentar aprender uma determinada função que "poderia" gera-los. Formalmente, isto significa que, dados exemplos de pares (x_i, $f(x_i)$), em que x_i é a entrada e $f(x_i)$ é a saída da função aplicada a x_i, então a tarefa é encontrar, dentre uma coleção de exemplos de f, uma função h que mais se aproxime de f. Esses métodos são apropriados quando existe alguma espécie de "professor" fornecendo os valores corretos para a saída da função de avaliação. Entretanto, se não houver nenhum "professor" fornecendo exemplos, o que o agente poderá fazer? É em cima deste contexto que o paradigma de Aprendizagem por Reforço (AR) torna-se um tema de grande interesse na área da AM, visto que dispensa um "professor" para nortear o processo de aprendizagem do agente. Este fato torna a AR

particularmente adequada para domínios em que a obtenção de exemplos de treinamento mostra-se difícil ou até mesmo impossível.

Na AR um agente aprende por sucessivas interações em um ambiente dinâmico [56]. Ele é responsável por selecionar possíveis ações para uma determinada situação apresentada pelo ambiente [57]. Por esse motivo os agentes da AR são caracterizados como autônomos. A questão que envolve a AR é basicamente: como um agente autônomo que atua sobre um determinado ambiente pode aprender a escolher suas ações para alcançar seus objetivos? Este é um problema muito comum em tarefas como o controle de um robô remoto e aprender a jogar jogos de tabuleiros, como Damas. O agente atua sobre o ambiente recebendo sinais (reforço ou penalidade), através de uma função de recompensa, para definir a qualidade da sequência de ações [58]. Por exemplo, em um jogo de tabuleiro como Damas, quando o agente consegue uma vitória ele receberá uma recompensa positiva (maior que zero, por exemplo), caso perca o jogo sua recompensa será negativa (menor que zero), mas se empatar sua atuação será neutralizada (recompensa igual a zero, por exemplo) [5].

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

A Figura 17 apresenta um sistema típico de AR. Basicamente, ele é composto de um agente interagindo em um ambiente via percepção e ação. A cada iteração t com o ambiente, o agente percebe o estado S_t - presente em um conjunto discreto de estados S - em que está inserido (pelo menos parcialmente) e seleciona uma ação A_t dentre o seu conjunto de ações A a ser executada em consequência de sua percepção. A ação executada muda, de alguma forma, o ambiente; e as mudanças são comunicadas ao agente por um sinal de reforço (R_t) [2].

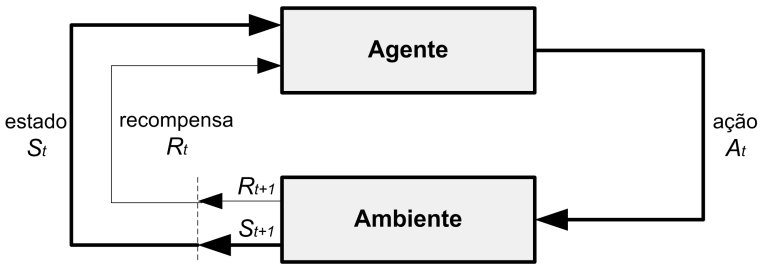


Figura 17 – Iteração do agente com o ambiente na aprendizagem por reforço [2]

O objetivo do método de AR é encontrar uma política, definida como um mapeamento de estados em ações, que maximize as medidas de reforço acumuladas ao longo do tempo.

Dentre os algoritmos existentes para solucionar o problema de AR, os métodos das Diferenças Temporais de Sutton (do inglês *Temporal Difference* (λ) - 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 [59]. 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 *Temporal Difference* (TD)(λ) são guiados pelo erro ou diferença entre predições sucessivas temporárias de estados sequenciais experimentados por um agente em um domínio, resultante de uma sequência de ações $(a_0, ..., a_{i-1}, a_i, a_{i+1}, ..., a_j)$ que são executadas ao longo do tempo com o objetivo de realizar uma tarefa f (aqui chamada de *fim-de-episódio*) para o qual 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[18].

Supondo que um agente execute, sobre um determinado domínio, uma sequência de ações $\{a_0, ..., a_{i-1}, a_i, a_{i+1}, ..., a_j\}$ ao longo do tempo, em que a ação a_j 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_j, 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 (por exemplo, no caso de Damas, se o agente perder). Resumindo, as predições de *fim-de-episódio*, isto é, P_j, são predições que representam reforços positivos ou negativos retornados pelo ambiente em função do desempenho final do agente no *fim-de-episódio*. Para todos os outros estados anteriores ao *fim-de-episódio*, em que nenhuma recompensa é retornada pelo ambiente, é a própria função de avaliação que define o agente que calcula a predição P_i associada ao estado resultante da ação executada a_i, em que $i \in \{0, 1, ..., i-1, i, i+1, ..., f-1\}$. 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(λ) 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(λ) usa essa informação retornada pelo ambiente para ajustar a função de avaliação do agente com base na diferença obtida pelas duas predições sucessivas [18].

O comportamento dos métodos TD(λ) faz com que sejam adequados para serem utili-

zados em jogos - como Gamão, Xadrez, Damas, dentre outros - em que 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.

Nos jogos de Damas (foco deste trabalho), as sequências de ações correspondem aos movimentos executados pelo agente ao longo dos jogos e as predições correspondem às avaliações realizadas em função do estado atual do jogo.

2.8 Técnica de Treinamento por Self-play com Clonagem

A ideia básica da técnica de treinamento por *self-play* com clonagem é treinar um agente através de vários confrontos contra uma cópia (clone) de si mesmo. Durante estes confrontos, no processo de treinamento do agente (não clonado), sucessivos ajustes são feitos neste agente de modo a acumular melhoria em sua performance. Caso tal agente apresente melhor performance em relação ao seu clone, uma nova cópia do agente ajustado é gerada. A partir deste momento, o agente passa a treinar com este novo clone. O processo é ilustrado na Figura 18 e detalhado na sequência.

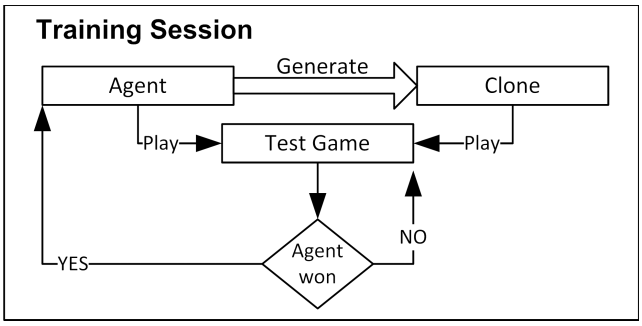


Figura 18 – Processo de treinamento por *self-play* com clonagem

Este processo é constituído por um número q_n de seções de treinamento composta por x jogos cada uma. Em cada seção q_i, o passo inicial é a geração de um agente ag com definição de parâmetros arbitrários, bem como sua cópia idêntica (clone) c_{ag}. Estes agentes irão disputar um jogo $x^{'}$, em que somente o agente ag terá seus parâmetros ajustados. No final do jogo, se ag for o vencedor, uma nova cópia $c^{'}_{ag}$ é gerada (a cópia gerada previamente é descartada) e um novo jogo é iniciado. Caso ag não vença, ele irá iniciar um novo jogo contra c_{ag}, a fim de continuar o processo de ajuste de seus parâmetros. Este processo é repetido durante x jogos. Quando estes jogos finalizam, a seção q_i termina, o melhor agente é armazenado e uma nova seção q_{i+1} é iniciada. Quando todas as seções estiverem concluídas, $(q_i == q_n)$, o melhor dos agentes de cada seção disputam um torneio. O vencedor torna-se o agente treinado que será utilizado

nas partidas reais (de não treino). Os agentes de Damas que estão relacionados entre os objetivos deste trabalho utilizam esta técnica no processo de aprendizado.

Capítulo 3

Estado da Arte

Como este trabalho de doutorado visa a construção de uma nova proposta de distribuição do algoritmo de busca Alfa-Beta e a produção de um sistema automático multiagente jogador de Damas para validar a proposta do algoritmo, os trabalhos correlacionados são apresentados em duas seções. A primeira, seção 3.1, relaciona os principais algoritmos de busca baseados no Alfa-Beta que atuam em ambiente de alto desempenho. A segunda, seção 3.2, apresenta os principais agentes automáticos jogadores de Damas propostos na literatura.

3.1 Algoritmos de Busca Paralelos baseados no Alfa-Beta

As versões paralelas do algoritmo Alfa-Beta tem por objetivo proporcionar um melhor *look-ahead* aos agentes jogadores que o utilizam como meio de tomada de decisão. Todavia, conforme apresentado na seção 2.4.2, estas versões precisam lidar com algumas fontes de ineficiência. Por esta razão, diversas versões paralelas do Alfa-Beta tem sido propostas na literatura segundo duas abordagens de paralelismo: síncrona ou assíncrona. As subseções 3.1.1 e 3.1.2 relacionam as principais versões paralelas do Alfa-Beta agrupadas segundo a abordagem de paralelismo adotada.

3.1.1 Algoritmos com Abordagem Síncrona de Paralelismo

Esta seção resume alguns dos principais algoritmos baseados na abordagem síncrona de paralelização do Alfa-Beta encontrados na literatura. Particularmente, o YBWC [34], por ser o mais eficiente dentre as versões síncronas do Alfa-Beta e por ser usado nos testes aplicados no presente trabalho, será apresentado em detalhes na seção 3.1.1.3.

3.1.1.1 PVS

O algoritmo conhecido como PVS proposto por [30] foi resultado das primeiras tentativas de paralelizar o Alfa-Beta e foi fonte de inspiração para vários dos métodos modernos. No PVS, o ramo mais a esquerda (variação principal) deve ser pesquisado antes de iniciar a busca paralela nos demais ramos. Todos os processadores percorrem o primeiro ramo de cada nó PV (ou tipo 1, conforme apresentado na seção 2.3.2.3) até alcançar o nó PV que é um nível acima dos nós folhas. Neste algoritmo, um processador pesquisa o ramo mais a esquerda, enquanto os outros ficam ociosos. Quando esta exploração termina, os demais processadores iniciam a busca nos demais ramos.

A Figura 19 ilustra o progresso de dois processadores, P_1 e P_2, ao explorar uma pequena árvore de busca utilizando PVS. Ambos os processadores percorrem o caminho mais à esquerda até chegarem ao nó d. Neste nó, P_2 permanece ocioso enquanto P_1 explora o ramo g. Uma vez finalizada a exploração de g, a busca paralela é iniciada: P_1 explora h enquanto P_2 explora i. Quando d tiver sido completamente avaliado, a busca é movida para o seu pai, a. Desta forma é iniciada a busca nos irmãos de d, onde o processador P_1 avalia e e o P_2 avalia f. Uma vez que a tiver sido completamente avaliado, a busca paralela volta à raiz da árvore com P_1 avaliando b e P_2 avaliando c [48]. Apesar de sua importância histórica, o PVS tem uma baixa eficiência quando testado em ambientes com um grande número de processadores.

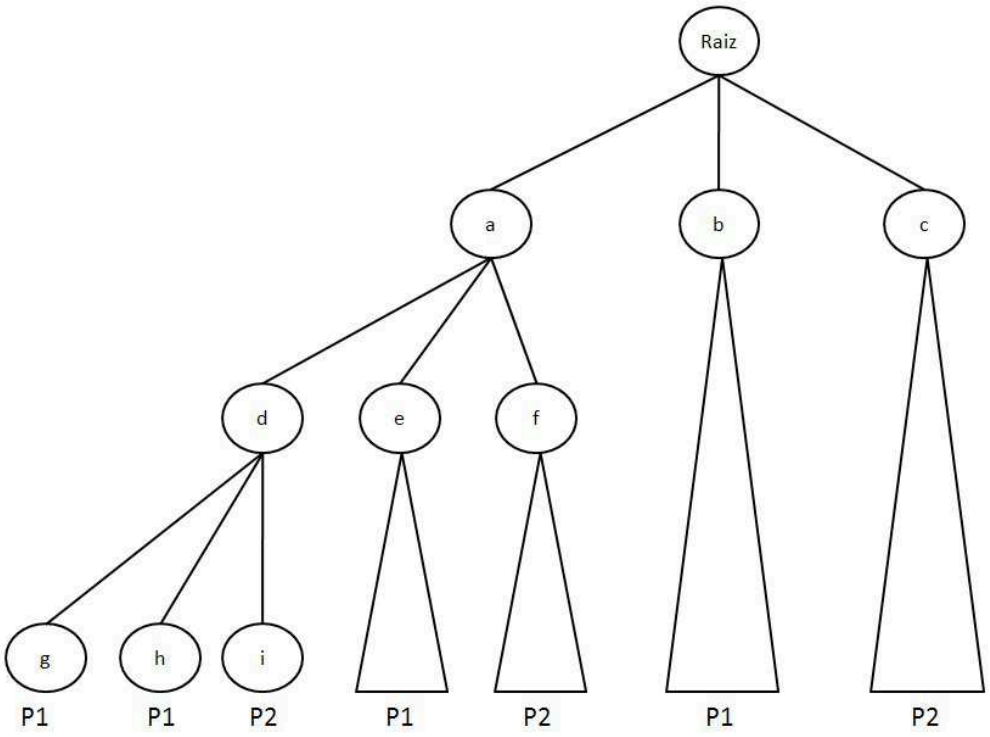


Figura 19 – Exploração paralela de uma árvore com 2 processadores pelo PVS

3.1.1.2 DTS

O algoritmo *Dynamic Tree Splitting* (DTS) [31] tenta predizer uma árvore mínima. Para isso, ele mantem uma avaliação de confiança para cada nó, indicando o grau de vizinhança que todos os filhos tem para serem explorados. O DTS atualiza a avaliação de confiança de um nó a cada vez que o tempo de busca de um filho finaliza. Esta atualização pode ser considerada como uma atualização da árvore mínima. No entanto, a atualização é realizada somente quando a busca de um filho é finalizada. Resultados de uma busca rasa em nós filhos não são utilizados [13].

O DTS utiliza um modelo de comunicação entre os processadores P2P (*pear-to-pear*). A responsabilidade de um nó tem uma abordagem diferente: enquanto diversos processadores podem colaborar na exploração de um nó, o processador que termina a sua porção de trabalho por último é o responsável por retornar o resultado da avaliação do nó ao seu pai [48].

Se um processador identifica um estreitamento da janela de busca, esta nova informação é compartilhada com os demais processadores que compartilham da busca do nó relacionado a este estreitamento. Se uma poda é descoberta, os processadores afetados por tal poda voltam para o estado ocioso e podem requisitar novas tarefas.

O DTS foi projetado para rodar em arquitetura de memória compartilhada. Por esta razão, experimentos com grande número de processadores não estão disponíveis, visto que, máquinas com um grande número processadores e com memória compartilhada são difíceis de serem obtidas [48].

3.1.1.3 YBWC

O *Young Brothers Wait Concept* (YBWC) é uma versão paralela síncrona do Alfa-Beta. O modelo de comunicação adotado é o mestre-escravo, onde um processador pode atuar como mestre e/ou escravo durante o processo de busca. A Figura 20 ilustra essa iteração entre os processadores.

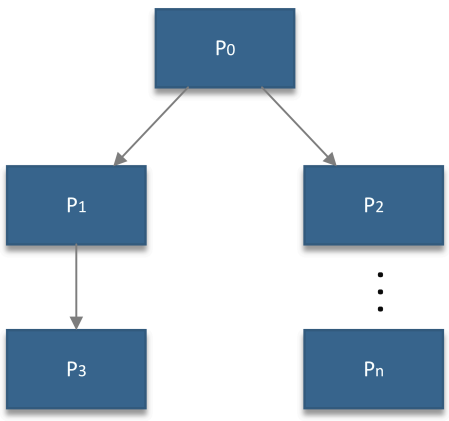


Figura 20 – Estrutura de comunicação formada entre os processadores no YBWC

Conforme é possível observar na Figura, o processador P_1 atua como escravo de P_0 e como mestre de P_3. O processador que não possui um mestre, na Figura, P_0, é denominado *Processador Principal* e é o responsável por retornar a solução final da busca. A iteração entre os processadores é feita através de mensagens de requisição. Desta forma, um processador que esteja em condições de explorar uma nova tarefa, isto é, que esteja *ocioso*, deve solicitar tarefas aos demais. Caso algum possua tarefa disponível, ele a envia. No YBWC, cada processador dispõe de uma pilha individual que é usada para manter o controle das tarefas.

A fim de ilustrar o funcionamento do YBWC, foi considerado o seguinte cenário:

- ❏ Um conjunto P de processadores definido como: $P = \{P_0, P_1, P_2, P_3\}$.
- ❏ O nó N_1 que deve ser explorado pelo algoritmo até a profundidade de busca d.
- ❏ O processamento de N_1 por P_0.

A Figura 21(a) ilustra a exploração de N_1 utilizando o YBWC. A Figura 21(b) ilustra a estrutura de controle dos nós (pilha) do processador P_0.

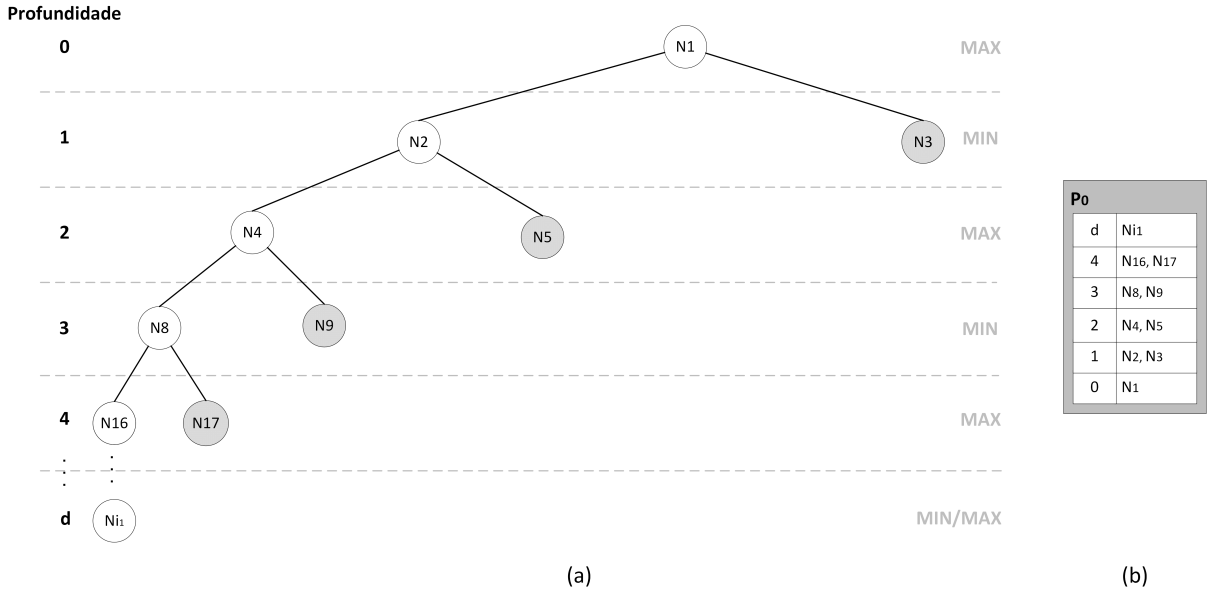


Figura 21 – (a) Árvore de exemplo (b) Ilustração da estrutura de controle de nós (pilha) do processador P_0 na expansão do nó N_1

Quando P_0 recebe N_1 para exploração, ele registra tal nó na sua pilha local. Na sequência, ele inicia a exploração do ramo mais à esquerda (*eldest brother*) de N_1 utilizando uma janela de busca com largura máxima $[-\infty, \infty]$. Esta expansão será realizada até a profundidade de busca d, sendo que todos os nós que forem gerados ao longo desse

processo serão alocados na pilha local de P_0. Ao final dessa expansão, o valor calculado para o nó mais profundo do ramo mais a esquerda (N_{i1}) será usado para estreitar a janela de busca (tal qual no Alfa-Beta). Somente então será disparada a distribuição, a ser efetuada por P_0, dos demais nós da árvore de N_1 (nós "*young brothers*" armazenados na pilha de P_0) entre os processadores que tenham requisitado tarefa. A janela inicial utilizada no começo do processamento desses nós distribuídos corresponde àquela estreitada produzida pelo processamento do "*eldest brother*" (N_1).

P_0 efetua a escolha do nó a ser distribuído a um escravo requisitante selecionando aquele localizado mais à esquerda e na maior profundidade da árvore de busca, pois, considerando que foi empregada a ordenação no modo *best-to-worst*, os nós localizados mais a esquerda tem mais chances de definir uma janela de busca que aumente o número de podas. Como o YBWC considera a profundidade de busca para distribuir suas tarefas, uma boa estratégia é alocar os nós na estrutura de pilha de acordo com sua profundidade. Na Figura 21(b) é possível observar que os nós estão armazenados no nível correspondente à sua profundidade na árvore de busca.

A iteração de requisição de tarefas entre os processadores é realizada através de mensagens. Um processador que deseje solicitar uma tarefa, seleciona algum processador no conjunto P e envia uma mensagem de requisição. A política de seleção do processador que receberá a requisição de tarefa é definida na implementação do algoritmo. Por exemplo, [60] segue uma política de escolha aleatória.

Um processador que recebe uma mensagem de requisição de tarefas verifica, em sua pilha local, se há alguma tarefa disponível. Caso positivo, a tarefa é enviada como resposta. Caso negativo, é enviada uma mensagem informando que não há tarefas disponíveis. O esquema ilustrado na Figura 22 dá uma visão geral deste processo ocorrendo após a exploração do nó N_{i1} por P_0.

Na Figura 22, inicialmente, P_1 solicita tarefas a P_2. Como P_2 não tem tarefas para compartilhar, ele envia uma mensagem de resposta *SEM_TAREFAS* a P_1. Nesta situação, P_1 envia uma nova mensagem de requisição de tarefa, desta vez a P_0. Este último verifica na sua pilha local se há alguma tarefa disponível para processamento. Na Figura, os nós representados pela cor *verde* estão disponíveis, ao passo que aqueles representados pela cor *laranja* estão em execução e os de cor *cinza* já tiveram sua exploração concluída. Neste contexto, P_0 seleciona o nó N_9 e o envia como resposta a P_1, que inicia a sua exploração.

Assim como P_1, os processadores P_2 e P_3 também enviam requisições de tarefas aos demais. Considerando que apenas P_0 tenha neste instante tarefas disponíveis, uma possível distribuição que pode ocorrer após a exploração de N_{i1} por P_0 (ilustrado na árvore da Figura 22) é ilustrada na Figura 23.

No YBWC, um processador que recebe uma tarefa t, torna-se responsável por ela até o fim, não podendo redistribui-la. Por exemplo, considerando o exemplo da Figura 23, a

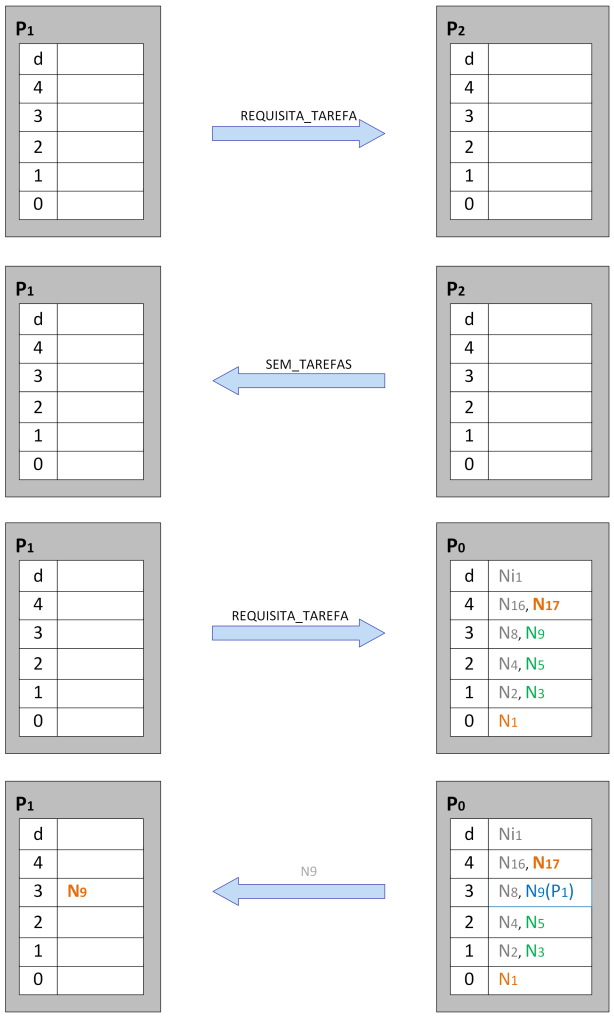


Figura 22 – Ilustração das requisições de tarefas efetuadas pelo processador P_1

tarefa N_9, atribuída a P_1, não pode ser transferida a outro processador. Neste caso, apenas as sub-tarefas de N_9 poderão ser distribuídas a outros processadores seguindo a mesma dinâmica feita inicialmente por P_0 ao iniciar a exploração de N_1, isto é, primeiramente, o ramo mais a esquerda é explorado. Devido ao fato de que, no YBWC, um processador estará apto a requisitar novas tarefas apenas após a conclusão da tarefa pela qual está responsável, um processador pode ficar ocioso aguardando por retornos vindos de seus processadores escravos. Por exemplo, na situação ilustrada na Figura 21, P_0 mesmo tendo concluído a exploração de N_{17}, manter-se-á ocioso até que seus escravos P_1, P_2 e P_3 lhes enviem os resultados do processamento de N_9, N_5 e N_3, respectivamente, uma vez que esses últimos nós integram o processamento do nó N_1 alocado a P_0. Tal ócio se deve à natureza intrinsecamente síncrona do YBWC.

A janela de busca é atualizada sempre que um processador identifica um estreitamento nos limites de sua janela atual. Neste caso, o processador que identificou o estreitamento

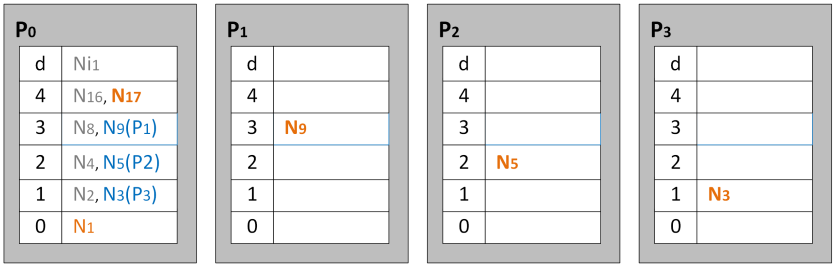


Figura 23 – Possível distribuição de tarefas em determinado momento a partir do YBWC.

informa tal fato ao seu mestre de forma que este último repasse os novos limites a seus demais escravos. Desta forma, os novos limites de busca são propagados a todos os processadores ativos. Por exemplo, ainda considerando a possível distribuição da Figura 23, suponha-se que P_1 tenha identificado um estreitamento de janela ao processar N_9. Neste momento, ele irá informar a P_0 os novos limites. P_0, então, enviará os novos limites a P_2 e P_3. A atualização destes novos limites em tais processadores pode ser bastante promissora para o processo de podas. De fato, eles podem possibilitar, por exemplo, que o processador P_2, executando N_5, identifique que um dado sucessor $N_5^{'}$ de N_5 não precisa ser explorado. Neste caso, será necessário avaliar os seguintes casos:

- Se P_2 tiver iniciado a exploração de $N_5^{'}$, ele abortará este processo imediatamente.

- Se $N_5^{'}$ estiver de posse de outro processador, P_2 enviará uma mensagem de *PODA*, para que tal processador interrompa sua exploração. Se a exploração de $N_5^{'}$ por tal processador tiver sido iniciada, ela será abortada imediatamente. Caso contrário, a exploração não ocorrerá.

Este esquema de atualização de janela de busca permite que o YBWC execute o mais próximo possível de como a versão serial do Alfa-Beta executaria. Sendo assim, em uma árvore ordenada, os limites da janela de busca será no melhor caso, iguais aos limites praticados pela versão serial. Tal fato permite que o YBWC retorne a mesma solução que a versão serial retornaria.

A fim de aprimorar o desempenho da busca devido à ocorrência de transposição de estados (seção 2.5.3), o YBWC utiliza TT. Além disso, o uso de TT torna-se muito adequado para o emprego da estratégia de busca por ID (seção 2.2.2). Em arquiteturas de memória distribuída, cada processador terá a sua própria TT, ou seja, é utilizado o conceito de TT distribuída apresentado na seção 2.4.3. Nesta situação, não há compartilhamento das informações entre as TTs, uma vez que mante-las sincronizadas tem um custo proibitivo devido a necessidade de tráfego constante dos dados pela rede.

Apesar das conveniências citadas na seção 2.2.2 da busca baseada em ID, o emprego do YBWC em arquitetura de memória distribuída pode fazer com que o algoritmo não usufrua totalmente das vantagens da utilização das TTs. Isso ocorre pelo seguinte motivo: na busca baseada em ID, a cada iteração o algoritmo executa uma busca em profundidade fixa limitada ao nível i prevista para aquela iteração, sendo que, de uma iteração a outra subsequente, tal nível é acrescido em 1 (um) [10]. Consequentemente, durante uma expansão por ID, há frequentes repetições de estados ao longo das iterações (fato que justifica, por sinal, o uso de TTs). No entanto, não existe garantias que em uma nova iteração $i + 1$, os processadores vão explorar as mesmas tarefas que exploraram na iteração i. Tal situação é inerente à forma com que o YBWC, opera que é por roubo de trabalho (*working-stealing*), visto que, se um processador P_j requisitar uma tarefa a um processador P_k e este último tiver uma tarefa disponível em sua pilha local, ele irá envia-la como retorno à solicitação de P_j. Neste contexto, pode ocorrer de, na iteração $i + 1$, a requisição realizada por P_j não ser direcionada a P_k. Tal fato, faz com exista a possibilidade da busca perder o benefício provido pelas informações gravadas na TT local dos processadores. Além disso, se ocorrer situações de transposição de estados, não será possível evitar o reprocessamento dos mesmos.

3.1.1.4 Alfa-Beta Paralelo Baseado em GPU

Em Ref. [61], os autores criaram uma versão paralela do Alfa-Beta sobre uma *Graphics Processing Unit* (GPU). Eles compararam a versão paralela com a versão serial do algoritmo utilizando o problema do jogo Othello. A implementação foi baseada no algoritmo PVS (seção 3.1.1.1). Como a versão do GPU utilizada pelos autores não suporta recursão, a versão criada foi implementada de modo iterativo usando controle manual de estruturas de pilhas em arquitetura de memória compartilhada. As partes do algoritmo Alfa-Beta que são executadas em paralelo são as *threads* referentes a avaliação dos nós, geração dos sucessores (movimentos), e execução do movimento. As demais tarefas tais como verificação de podas e atualização de avaliações ficaram sob a responsabilidade de uma única *thread*.

Com os resultados obtidos, os autores conjecturaram que a aplicação desta estratégia de busca em problemas de maior complexidade como o Go pode ser apropriada.

3.1.1.5 Distribuição do Alfa-Beta com Política de Prioridades

Em Ref. [12], os autores propuseram uma versão paralela do Alfa-Beta em que empregaram uma política de divisão de tarefas denominada *two-level task scheduling policy*. Nesta política, todos os nós são classificados por uma das seguintes prioridades:

Alta: nó raíz; o sucessor localizado mais a esquerda de um nó de alta prioridade; sucessores de um nó do tipo PV (tipo 1) ou ALL (tipo 3);

Baixa: sucessores de um nó de baixa prioridade ou que o pai seja do tipo CUT (tipo 2).

Este algoritmo trabalha empregando o modelo de paralelismo mestre-escravo. A comunicação entre os processadores é feita apenas entre mestre e escravos, não há comunicação entre os escravos.

Para uma busca com profundidade máxima igual a d, primeiramente, o mestre expande a árvore até uma profundidade $d-g$, em que g é denominado granularidade e representa a profundidade de busca em que os escravos irão executar suas tarefas. Os nós contidos no limite $d-g$ são nós folhas. A expansão realizada pelo mestre utiliza o algoritmo Alfa-Beta serial em conjunto com TT e ID para ordenação dos movimentos. Assim, os nós folhas são enviados aos escravos de acordo com sua prioridade.

A distribuição de tarefas aos escravos é feita por rodízio, isto é, a partir do modelo *round-robin*. Cada escravo possui a sua própria TT, que é limpa apenas quando o mestre conclui a busca. Ao receber uma tarefa, um escravo a explora até a profundidade g. Cada escravo recebe uma tarefa por vez a menos que receba uma de alta prioridade. Neste caso, se o escravo estiver explorando a tarefa de baixa prioridade ele a suspenderá imediatamente e iniciará a exploração da tarefa de alta prioridade. Quando finalizar, se não houver uma nova tarefa de alta prioridade, ele reinicia a exploração da tarefa suspensa. Como as informações contidas na TT são retidas durante a suspensão da tarefa, a degradação de performance é pequena.

Quando um escravo finaliza a exploração de um nó, ele envia as informações ao mestre. O mestre, por sua vez, ao receber estas informações realiza uma série de atualizações que incluem: ajuste do valor da avaliação do nó folha em questão, podas de subárvores que se tornaram desnecessárias à busca, estreitamento da janela de busca, e promoção de nós de baixo nível. Se o mestre precisar atualizar informações em alguma tarefa que já tenha sido enviada a algum escravo, ele envia mensagens de atualização a tal escravos.

A avaliação deste algoritmo foi feita em um forte jogador de *Soghi* (xadrez japonês). Na avaliação dos autores a política de prioridades diminuiu a exploração de nós desnecessários (sobrecarga de busca) comparado à versão do algoritmo YBWC (a ser apresentado na seção 3.1.1.3) e apresentou melhor performance em árvores de jogo mais profundas (de maior granularidade).

3.1.2 Algoritmos com Abordagem Assíncrona de Paralelismo

Esta seção apresenta os principais algoritmos baseados na abordagem assíncrona de paralelização do Alfa-Beta encontrados na literatura. Salienta-se que o algoritmo APHID, por fazer parte do cumprimento dos objetivos deste trabalho, será apresentado com maiores detalhes na seção 3.1.2.2.

3.1.2.1 UIDPABS

O UIDPABS [32] foi a primeira tentativa de paralelizar o Alfa-Beta seguindo a abordagem assíncrona de paralelismo. Neste algoritmo, os estados filhos da raiz da árvore são divididos entre os processadores. Tais processadores realizam a busca das tarefas que lhe são atribuídas utilizando ID. Cada processador possui a mesma janela de busca inicialmente, mas suas janelas podem ser modificadas baseadas nos resultados de busca de suas tarefas. O UIDPABS combina os resultados assim que um tempo limite é atingido. Alguns estados podem ter sido explorados em níveis mais profundos do que outros nós que foram explorados por outros processadores [13]. A Figura 24 ilustra o funcionamento geral do UIDPABS.

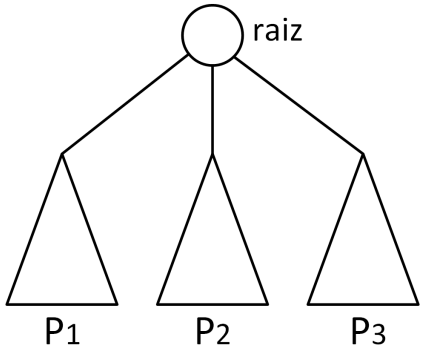


Figura 24 – Exploração paralela de uma árvore com 3 processadores pelo UIDPABS

3.1.2.2 APHID

O algoritmo *Asynchronous Parallel Hierarquical Iterative Deepening* (APHID) é uma versão paralela do Alfa-Beta que segue a abordagem assíncrona de paralelismo a partir do modelo de comunicação mestre-escravo entre os processadores. Em um conjunto P de processadores, apenas um atuará como mestre e os demais atuarão como escravos, conforme ilustrado na Figura 25, em que P_0 é o mestre e os demais são escravos.

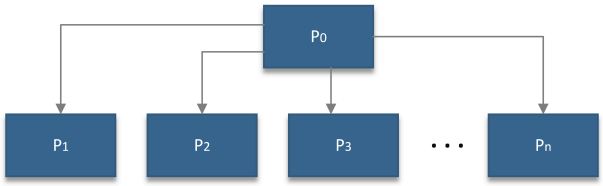


Figura 25 – Estrutura de comunicação entre os processadores no APHID

O processador mestre é o responsável por gerenciar a busca e retornar a solução final. Os processadores escravos, por sua vez, processam os nós que lhes foram repassados pelo mestre segundo a estratégia de busca ID (seção 2.2.2) e, uma vez concluído o processamento, retornam ao mestre o resultado de suas explorações. Para que o uso de ID e

a ocorrência de transposição de estados (seção 2.5.3) não impactem na performance do algoritmo, são utilizadas TTs. No APHID, em arquiteturas de memória distribuída, cada processador terá a sua própria TT e contará com um mecanismo que tenta amenizar as perdas dessa distribuição.

A fim de evidenciar as características dos processadores que atuam no APHID, as subseções 3.1.2.2 e 3.1.2.2 detalham as atividades desempenhadas por um processador mestre e por um escravo, respectivamente. Além disso, a seção 3.1.2.2 detalha como o APHID implementa TTs em uma arquitetura de memória distribuída. Com o intuito de auxiliar na compreensão do APHID, estas subseções consideram o seguinte cenário:

- ❏ Existe um conjunto P formado por quatro processadores $P = \{P_0, P_1, P_2, P_3\}$.
- ❏ O processador P_0 é o mestre e os demais são escravos.
- ❏ É apresentado ao algoritmo de busca um estado denominado N_1, que deve ser explorado até a profundidade máxima d.

Atividades do Processador Mestre

A árvore de busca apresentada na Figura 26 ilustra a exploração de N_1 por P_0.

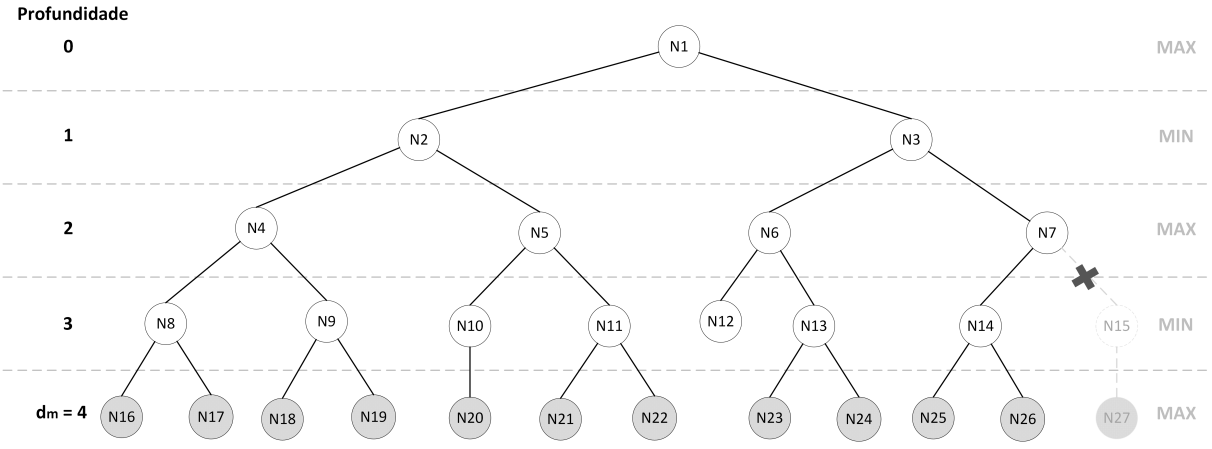


Figura 26 – Exemplo da árvore explorada pelo processador mestre no APHID

No APHID, P_0 efetuará a exploração de N_1 até a profundidade de busca d_m utilizando o algoritmo Alfa-Beta serial versão *fail soft*, onde $d_m < d$. Na Figura 26 d_m vale 4. Os nós da subárvore de P_0 são ordenados do modo *best-to-worst*, desta forma, nós que possuem maior probabilidade de ocasionar uma poda, ficarão localizados mais a esquerda da árvore de busca. Os nós existentes no nível d_m constituem uma *fronteira* de busca e serão distribuídos entre os escravos. Na Figura 26, os nós da fronteira estão apresentados em destaque. Observa-se no esquema que o nó N_{15} foi podado na busca. A exploração compreendida entre a raiz da árvore de busca e d_m é denominada *passagem* do mestre por

sua subárvore (a partir deste ponto, referida, simplesmente, por *passagem*). O processador mestre realizará diversas *passagens* até que a busca seja concluída a fim de atualizar as informações inerentes à fronteira. A profundidade d_m é definida empiricamente, todavia, deve possuir um valor que não comprometa a performance do algoritmo, visto que o mestre realizará o processo de passagem constantemente.

A janela de busca inicial utilizada nas *passagens* realizadas por P_0 possui abertura máxima, ou seja, alfa = $-\infty$ e beta = ∞. Cada vez que o mestre recebe informação de novos resultados providos de seus escravos, ele reinicia uma nova *passagem* (sempre com janela [$-\infty$ - ∞]), com base nesses novos valores. Conforme será apresentado a seguir, o efeito desses resultados retornados pode ter significativo impacto na dinâmica de podas da nova *passagem* a ser processada (lembrando que as *passagens* do mestre no APHID são sempre expansões seriais Alfa-Beta).

A exploração da subárvore de P_0 produz um valor minimax k (correspondente ao nó N_1), também denominado *valor hipotético*, que é enviado aos processadores escravos para formação e/ou atualização de suas janelas de busca, conforme detalhado na seção 3.1.2.2.

A distribuição de tarefas entre os escravos segue o método *round-robin* (cada folha é atribuída, sequencialmente, a um dos processadores). A Tabela 3 apresenta o resultado desta distribuição entre os processadores P_1, P_2 e P_3.

Tabela 3 – Divisão de tarefas segundo o modo *round-robin* entre três processadores.

Processador	**Tarefas**
P_1	N_{16}, N_{19}, N_{22}, N_{25}
P_2	N_{17}, N_{20}, N_{23}, N_{26}
P_3	N_{18}, N_{21}, N_{24}

Os valores da avaliação dos nós da fronteira de P_0 serão constantemente atualizados pelos resultados recebidos dos escravos que processaram as subárvores desses nós segundo a estratégia de busca por ID. É importante ressaltar que, devido à estratégia de busca, um escravo envia diversos resultados de avaliação de um mesmo nó ao mestre em momentos diferentes, isto é, cada avaliação corresponde a uma profundidade de busca em nível mais profundo. Desta forma, as novas *passagens* de P_0 por sua subárvore irão considerar os últimos valores recebidos dos escravos (os mais atuais) obtidos em profundidades maiores. Tal fato pode ocasionar as seguintes situações:

- Ao longo de uma *passagem*, P_0 pode obter um novo valor hipotético k' para N_1 que deverá ser enviado aos escravos para que eles atualizem suas janelas de busca (detalhes desta atualização serão apresentados na seção 3.1.2.2).

- Nós que existiam na fronteira de busca de uma dada *passagem* de P_0, podem ser podados em uma *passagem* subsequente. Por exemplo, caso o nó N_{11} seja podado em uma dada *passagem*, os nós N_{21} e N_{22} deixam de existir na fronteira de busca

de P_0. Como tais nós já haviam sido distribuídos a P_3 e P_1, respectivamente, P_0 irá informar a tais processadores que as tarefas N_{21} e N_{22} devem ter a prioridade alterada para a mínima (zero).

- ❏ Nós que haviam sido podados em uma *passagem* anterior de P_0 por sua subárvore, podem aparecer na fronteira de busca de uma *passagem* subsequente. Por exemplo, o nó N_{15}, que tinha sido podado na *passagem* mostrada na Figura 26, pode reaparecer em uma *passagem* posterior. Neste caso, haverá duas possibilidades: caso tal nó não tenha sido atribuído a um escravo previamente, ele será distribuído pelo mestre segundo o método *round-robin*; caso contrário, ele será mantido com o mesmo processador, mas com a prioridade reavaliada.

- ❏ A ordenação dos nós da fronteira de busca pode ser alterada.

A comunicação entre o mestre e os escravos é feita através de uma estrutura de dados denominada AphidTable disponível em cada processador. Basicamente, esta estrutura possui as seguintes informações: o identificador do processador, a lista de tarefas (nós) atribuídas a ele e o valor hipotético da última *passagem* do mestre por sua subárvore. Apesar de cada processador escravo possuir sua própria AphidTable, apenas P_0 possui uma cópia completa das tabelas dos escravos. A Figura 27 ilustra as estruturas AphidTable do mestre P_0 e dos escravos. Salienta-se que a aparição das tarefas nas AphidTables segue o ordem decrescente de prioridade das mesmas.

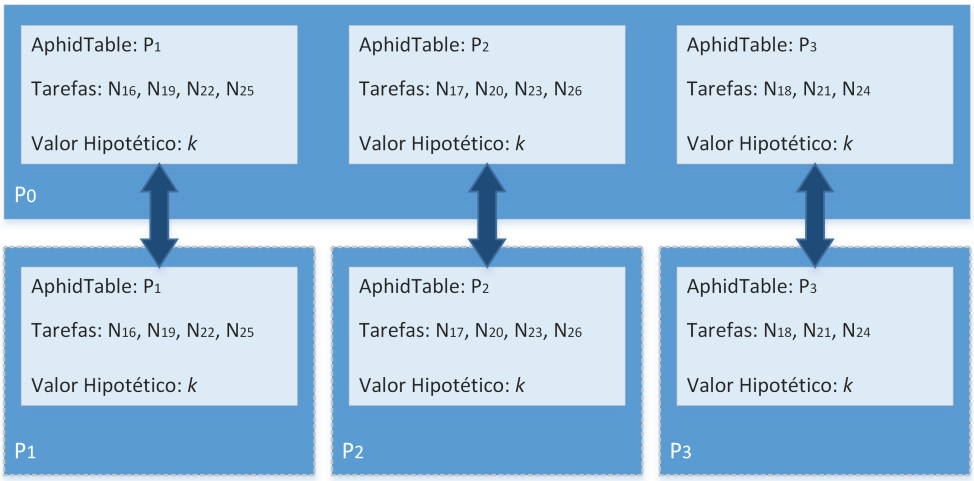


Figura 27 – Exemplo da estrutua AphidTable no processador mestre (P_0) e escravos (P_1, P_2 e P_3)

Atividades dos Processadores Escravos

No início do processo de busca, os processadores escravos aguardam o registro de tarefas em suas respectivas tabelas. Logo que uma tarefa é registrada, eles iniciam suas

atividades. Conforme apresentado na Tabela 3, cada escravo pode receber uma lista de tarefas. Desta forma, ele deve escolher uma tarefa para iniciar o processo de busca. A escolha de um nó em relação aos demais é baseado nos seguintes critérios de prioridade:

1. *Profundidade de busca*: os nós localizados em profundidade mais rasa tem maior prioridade, pois representam mais trabalho a ser realizado (lembrando que as profundidades de exploração dos nós da AphidTable vão variando ao longo da estratégia ID).

2. *Localização do nó na subárvore de busca do mestre*: como no APHID os filhos dos nós são ordenados do modo *best-to-worst*, explorar os nós localizados mais a esquerda pode aumentar o número de podas, beneficiando o processo de busca. A localização de um nó é informada pelo mestre.

3. *Participação na fronteira de busca do mestre*: a cada *passagem* do mestre por sua subárvore, a fronteira de busca pode sofrer alguma alteração. Desta forma, novos nós recebem prioridade maior devido ao critério 1. Por outro lado, os nós que não apareceram na *passagem* corrente do mestre por sua subárvore recebem prioridade zero.

Considerando o processador escravo P_1, sua lista de tarefas (N_{16}, N_{19}, N_{22} e N_{25}) e partindo do princípio de que estas tarefas acabaram de ser repassadas a P_1 por P_0, todas possuem a mesma profundidade de busca, logo, o critério 1 de prioridade não classifica tarefa alguma em relação às demais. Além disso, como tais tarefas foram obtidas pela primeira *passagem* do mestre, todas estão presentes na fronteira de P_0, portanto, o critério 3 também não pode ser aplicado. No entanto, a localização destes nós na árvore de busca é diferente e o critério 2 pode ser empregado. Assim, o nó localizado mais a esquerda corresponde à tarefa N_{16}, portanto, ela será selecionada para P_1 iniciar suas atividades. Quando nenhum dos critérios classifica as tarefas, a primeira da lista é a escolhida. É importante ressaltar que a prioridade das tarefas atribuídas aos processadores pode sofrer alterações no decorrer da busca, em virtude das novas *passagens* do mestre por sua subárvore, conforme explanado na seção 3.1.2.2.

Antes de iniciar a exploração de N_{16}, P_1 define a janela de busca inicial cujos limites são definidos a partir do valor hipotético enviado por P_0 e de um valor constante C - definido nas configurações do algoritmo - que irá estabelecer a largura da janela segundo a equação 6 para o limite inferior e pela equação 7 para o limite superior:

$$\alpha \leftarrow \textit{valor hipotético - } C; \tag{6}$$

$$\beta \leftarrow \textit{valor hipotético + } C; \tag{7}$$

Ainda que a janela de busca de P_1 sofra um estreitamento no processamento de N_{16}, os novos limites não são repassados aos demais processadores, uma vez que no APHID não há atualização de janela entre os escravos (cada processador trabalha com sua própria janela de busca). Assim sendo, os escravos somente atualizam suas janelas em função do valor hipotético recebido do mestre. Considerando as tarefas de P_1, caso a exploração de N_{16} já tenha iniciado e um novo valor hipotético tenha sido registrado em sua AphidTable, tal valor apenas será considerado na definição de janela da próxima tarefa selecionada na lista de P_1, isto é, N_{16} não sofrerá o impacto desta atualização.

Uma vez definida a janela inicial de busca, P_1 inicia a exploração de N_{16} utilizando o algoritmo Alfa-Beta serial aliado à estratégia de busca por ID. Como os escravos exploram seus nós por ID, inicialmente P_1 irá explorar todos os nós na profundidade $d_m + i$, onde i equivale ao valor do incremento de cada iteração da busca. Considerando que $i = 1$, P_1 irá explorar N_{16} na profundidade $d^{'} = 5$. A Figura 28 ilustra a expansão da subárvore de N_{16} a partir da profundidade 5 (Figura 28(A)) até d (Figura 28(D)).

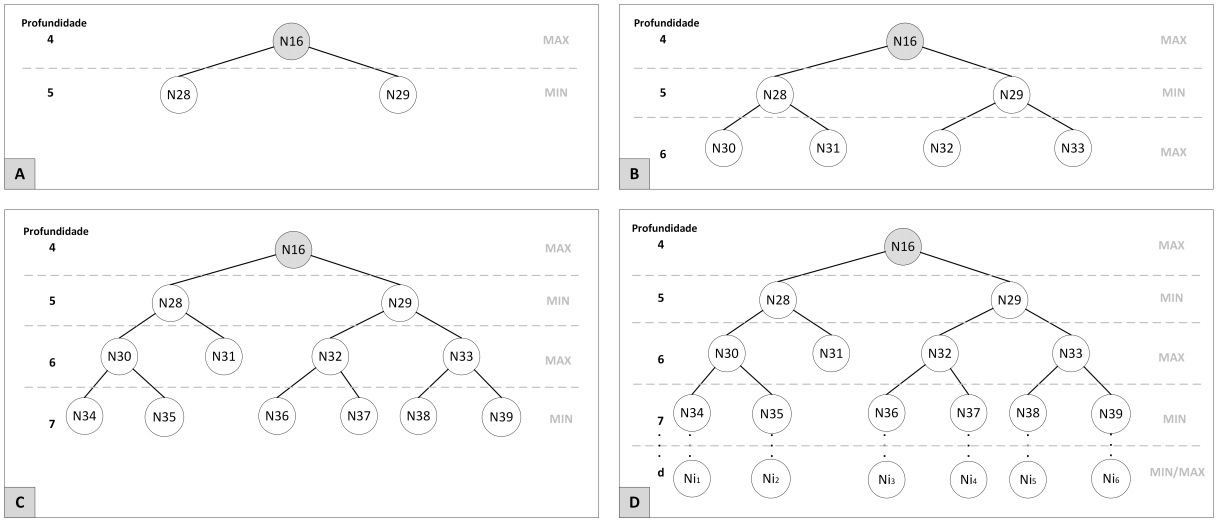


Figura 28 – Exemplo da exploração do nó N_{16} no processador escravo P_1 por ID considerando o valor do incremento igual a 1.

Tão logo P_1 finalize a exploração de N_{16} na profundidade 5 (Figura 28(A)), o novo valor obtido para tal nó será enviado ao processador mestre para atualização de sua fronteira de busca. Além disso, N_{16}, nesse momento, terá sua prioridade alterada na AphidTable de P_1 em função dessa nova profundidade 6 (Figura 28(B)). Assim sendo, não há garantias de que P_1 retomará a exploração de N_{16} imediatamente após finalizar sua exploração na profundidade 5, pois N_{16} pode não ser o nó de maior prioridade naquele momento - e os escravos sempre selecionam para exploração nós de maior prioridade em suas respectivas tabelas. Em resumo, a profundidade de busca adotada para o nó selecionado corresponde à última exploração desta tarefa acrescida do valor do incremento de cada iteração da busca.

Esse processo irá se repetir até que todas as tarefas sejam exploradas na profundidade máxima d, conforme ilustrado na Figura 28.

No APHID, quando uma tarefa é atribuída a um escravo ela irá permanecer com ele até o final da busca, exceto no caso de balanceamento de carga, que é gerenciado pelo processador mestre. Neste caso, P_0 verifica se algum processador escravo já finalizou o processamento de todas as suas tarefas na profundidade d. Caso positivo, ele checará, ordenadamente dentre os demais escravos, o primeiro que está sobrecarregado, ou seja, que tem pelo menos um nó em sua AphidTable cujo processamento está em "compasso de espera". Encontrado esse primeiro escravo sobrecarregado, sua tarefa em "compasso de espera" de maior prioridade é repassada pelo mesmo ao escravo ocioso. Por exemplo, considerando a árvore de exploração de N_1 (Figura 26), caso P_2 esteja explorando o nó N_{17} na profundidade $d^1 = 6$ e P_3 tenha finalizado a exploração dos nós N_{18}, N_{21} e N_{24} na profundidade máxima de busca d, o mestre irá atribuir a P_3, a tarefa não finalizada de P_2 com maior prioridade, neste caso, N_{20}. Desta forma, N_{20} passa a pertencer ao processador P_3 e, caso ele já tenha sido explorado em profundidades mais rasas, tais informações serão descartadas e ele será reexplorado. Uma vez que N_{20} é atribuído a P_3, ele não poderá ser realocado novamente durante o processo de busca, isto é, no APHID um nó pode ser redistribuído por balanceamento de carga apenas uma **única** vez. Tal fato ocorre por duas razões:

1. Para não sobrecarregar a rede com o tráfego de informações referentes a realocações constantes;

2. Para que a performance do algoritmo não seja prejudicada com o reprocessamento destes nós.

Tabela de Transposição

No APHID, caso o algoritmo seja implementado em arquitetura de memória distribuída, cada processador irá possuir sua própria TT. Neste caso, é impraticável manter todos os processadores totalmente atualizados em relação a todas as TT's devido à sobrecarga de comunicação que tal situação acarretaria. De fato, não haver a disponibilidade de todas as informações pode aumentar a sobrecarga de busca pelos seguintes fatores: utilização da estratégia de busca por ID; emprego do balanceamento de carga; e ocorrência de transposição de estados, como ocorre em problemas como o jogo de Damas (seção 2.5.3). O APHID tenta tratar cada um destes fatores da seguinte forma:

Estratégia de Busca por ID: o algoritmo mantém a atribuição de estados aos mesmos processadores durante toda a busca exceto os casos de balanceamento de carga, conforme explanado na seção 3.1.2.2.

Balanceamento de Carga ou Transposição de Estados: o algoritmo utiliza um mecanismo cíclico de compartilhamento entre processadores pares - processadores escravos que possuem o mesmo mestre - através de uma extensão da TT local denominada Tabela de Transposição Sombra (TTS). Este mecanismo é transparente à busca, de modo a não impactar o fluxo normal do algoritmo APHID. A subseção 3.1.2.2 apresenta detalhes deste mecanismo.

Tabela de Transposição Sombra

O mecanismo de compartilhamento entre os processadores escravos no APHID em arquitetura de memória distribuída parte do princípio de que processadores que possuem um mesmo mestre, são pares. Neste contexto, o APHID permite a comunicação unidirecional entre uma lista de processadores pares a fim de empregar um meio de aprimorar a utilização de TT em arquitetura de memória distribuída. Para isso, utiliza-se uma extensão da TT local de cada processador denominada TTS. Esta tabela possui tamanho menor do que a TT local visando tornar o seu compartilhamento rápido e não sobrecarregar a rede. A existência da TTS é transparente para a aplicação. Apenas quando a TTS atingir o seu limite poderá ser compartilhada. O tamanho da TTS é prefixado nas configurações do sistema. Quando o seu limite é atingido, ela é enviada para o processador par a sua direita na lista de escravos e apagada no processador de origem a fim de que novos estados possam ser armazenados para um novo ciclo. O processador que recebe uma TTS atualiza sua TT local e repassa a TTS recebida ao próximo processador à sua direita. A Figura 3.1.2.2 ilustra um exemplo deste mecanismo.

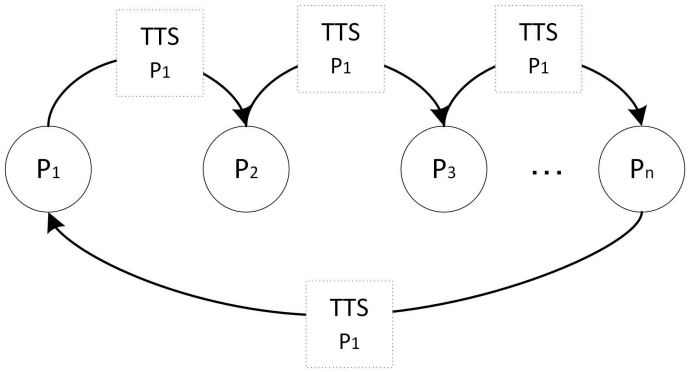


Figura 29 – Exemplo do compartilhamento de TTS entre os processadores em uma arquitetura de memória distribuída

Conforme pode ser observado, os processadores P_1, P_2, P_3, P_n se comunicam apenas com seu par (vizinho) mais próximo a direita. Caso o processador seja o último da fila, ele comunica-se com o primeiro formando assim, um ciclo. Quando o processador que iniciou o ciclo recebe a TTS que compartilhou, ele estará apto a iniciar um novo ciclo.

Cada processador possui seu próprio ciclo de compartilhamento. Apenas é permitido um ciclo por vez por processador.

O tempo de compartilhamento de uma TTS é crítico, visto que se o envio de uma TTS demorar muito para atingir determinado escravo, estados existentes dentro desta TTS poderão ter sido reprocessados. Então, a utilidade dos dados de uma TTS diminui a medida que o tempo de compartilhamento aumenta. Portanto, é importantíssimo ter uma TTS que possa ser enviada a todos os processadores o mais rápido possível.

Neste sentido, o tamanho da TTS é um fator muito importante a ser considerado. Quanto maior o número de entradas na TTS, maior o tempo gasto para o envio a cada escravo. É preferível uma TTS pequena e rápida do que uma TTS grande e lenta, devido ao tempo crítico que este processo demanda. Assim, todos os processadores terão a chance de ter uma TT com mais informações que poderão ser úteis na exploração de suas tarefas.

3.1.2.3 Distribuição do Alfa-Beta com Atualização Dinâmica de Subárvore

Em Ref. [13], os autores propuseram uma versão assíncrona paralela do Alfa-Beta, que adota o modelo de paralelismo adotado é mestre-escravo. Um processador mestre inicia a busca expandindo a árvore de jogo da raiz até uma profundidade de busca $d^{'}$. Os nós folhas dessa subárvore definem uma fronteira e serão distribuídos entre os demais processadores (escravos) que irão realizar uma busca por ID até a profundidade d (limite máximo da busca), onde $d > d^{'}$. A exploração dos nós nos escravos segue a política de prioridades proposta por [12]. A exploração realizada pelo mestre nesta subárvore ocorre diversas vezes durante o processo de busca e é denominado *passagem*. Estas *passagens* tem o intuito de atualizar a busca com os resultados recebidos dos escravos.

Os escravos enviam o resultado da exploração dos nós que lhe foram atribuídos de modo assíncrono. Para isso, existe uma verificação constante por parte do mestre para verificar atualizações em sua fronteira de busca. Neste sentido, os autores de [13] fizeram uma análise do efeito da atualização dos nós na fronteira da subárvore explorada pelo mestre, que pode ser feita de duas formas:

1. **Sem atualização dinâmica:** os escravos só enviam o resultado de suas tarefas ao mestre quando alcançam a profundidade máxima de busca. Desta forma, o mestre só atualizará a fronteira de sua subárvore com os resultados finais da avaliação de cada escravo.

2. **Com atualização dinâmica:** a medida que os escravos vão explorando suas tarefas por ID eles enviam ao mestre o resultado de suas avaliações mais rasas de modo que o mestre possa fazer uso destas informações nas suas novas *passagens* por sua subárvore.

As ideias deste algoritmo são baseadas no APHID (a ser detalhado na seção 3.1.2.2). Os experimentos foram realizados considerando estados gerados em árvores de jogos sinté-

ticas e estados obtidos a partir do programa *Soghi* (xadrez japonês). Os autores consideraram que a atualização dinâmica é importante para reduzir o tempo de busca. Ressalta-se a performance deste algoritmo não foi verificada em uma partida. Além disso, não foi realizado nenhum tipo de comparação com outros algoritmos paralelos, síncronos ou não.

3.2 Agentes Automáticos Jogadores de Damas

Devido à vasta disponibilidade de jogadores automáticos de Damas fundamentados nas mais diversas técnicas de aprendizagem, esta seção se divide em dois tópicos principais: jogadores com forte supervisão humana e jogadores com o mínimo de intervenção humana (aqui classificados como jogadores não supervisionados). No grupo dos jogadores supervisionados, este trabalho apresenta os dois principais jogadores, o Chinook [62] e o Cake [63]. No grupo dos jogadores não supervisionados apresenta-se os principais jogadores desenvolvidos pela equipe a qual o presente trabalho se insere, além de apresentar outros jogadores recentemente propostos, os quais possuem uma metodologia de aprendizagem diferente da utilizada pelos jogadores desta equipe.

3.2.1 Supervisionados

3.2.1.1 Chinook

O projeto Chinook foi iniciado em 1989 na tentativa de melhor entender as buscas heurísticas. O Chinook se tornou o mais famoso e mais forte jogador de Damas do mundo [62]. Ele é o campeão mundial homem-máquina para o jogo de Damas e conta com um conjunto de funções de avaliação ajustadas manualmente, ao longo de vários anos, para estimar quanto um determinado estado do tabuleiro é favorável para o jogador. Além disso, ele tem acesso a coleções de jogadas utilizadas por grandes mestres na fase inicial do jogo (*opening books*) e um conjunto de bases de dados para as fases finais do jogo com 39 trilhões de estados do tabuleiro (todos os estados com 10 peças ou menos) com valor teórico provado (vitória, derrota ou empate). Para escolher a melhor ação a ser executada, o jogador utiliza um procedimento de busca minimax distribuída com poda Alfa-Beta, aprofundamento iterativo e tabela de transposição.

O Chinook conseguiu, em 1994, o título de campeão mundial de Damas ao empatar 6 jogos com o Dr. Marion Tinsley que, até então, defendia seu título mundial há mais de 40 anos.

Em Damas, é fundamental evitar algumas posições no início de um jogo. Por isso a utilização de bases de dados de início de jogos (*openning books*) é bastante interessante, uma vez que um movimento inicial ruim pode fazer com que, antes do quarto movimento, seja encontrada uma situação de derrota forçada. A utilização de bases de dados de finais de jogos em Damas tem papel fundamental para a construção de agentes jogadores

de alto desempenho. No Chinook, essas bases contribuíram significativamente para seu sucesso contra os melhores humanos. Tais bases representam cerca de 400 bilhões de estados possíveis do tabuleiro [62]. Uma vez que as bases de dados são indispensáveis para o sucesso do jogador, um eficiente algoritmo de busca é utilizado para permitir rápida localização e execução da melhor ação.

Em 2007, a equipe do Chinook anunciou que o jogo de Damas estava fracamente resolvido (*weakly solved*), isto é, a partir da posição inicial padrão do jogo, existe uma prova computacional de que o jogo é um empate. A prova consiste em uma estratégia explícita com a qual o programa nunca perde, isto é, o programa pode alcançar o empate contra qualquer oponente jogando tanto com peças pretas quanto brancas [16].

3.2.1.2 Cake

O Cake está entre os jogadores de Damas mais competitivos do mundo. Este jogador é superior a versão do Chinook [62] que venceu o campeonato mundial contra Marion Tinsley [63]. Este jogador utiliza bases de dados de fim de jogo, a qual contém conhecimento perfeito do valor de qualquer movimento para tabuleiros com até 8 peças. Ele também utiliza um livro de abertura de jogos (*openning book*) que contém cerca de 2 milhões de movimentos. Para atuar no jogo entre as fases de inicio e fim de jogo (fase em que não possui acesso a BDs), ele usa o algoritmo de busca MTD(f) para escolher o próximo (melhor) movimento. Assim como o Chinook, o Cake é um jogador fortemente supervisionado

O jogador Cake está disponível através da plataforma *CheckerBoard*, que é a mais completa interface livre para agentes jogadores automáticos de Damas [64]. Esta plataforma suporta arquivos em formato PDN (*Portable Draughts Notation*), que é o padrão para 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.

3.2.2 Não Supervisionados

3.2.2.1 NeuroDraughts

O sistema NeuroDraughts da Lynch [65], [66] implementa o agente jogador de Damas como uma rede neural MLP que utiliza a busca Minimax para a escolha da melhor jogada em função do estado corrente do tabuleiro do jogo. Além disso, ele utiliza o métodos das diferenças temporais TD(λ) - (TD) - aliado à estratégia de treino por *self-play* com clonagem, como ferramentas para atualizar os pesos da rede. Para tanto, o tabuleiro é representado por um conjunto de funções que descrevem as características do próprio jogo de Damas (*features*). Particularmente, o NeuroDraughts fez uso de 12 características escolhidas manualmente dentre as 26 propostas no trabalho de Samuel [52]. A utiliza-

ção de um conjunto de características para representar o mapeamento do tabuleiro de Damas na entrada da rede neural é definida por Lynch como sendo um mapeamento NET-FEATUREMAP apresentado em detalhes na seção 2.5.2.2. O NeuroDraughts foi a base para a construção dos jogadores automáticos que compõem a linha de pesquisa que o presente trabalho se encaixa.

3.2.2.2 VisionDraughts

O VisionDraughts é um jogador de Damas automático proposto por Caixeta e Julia [51] cuja arquitetura é inspirada no NeuroDraughts. Todavia, o mecanismo de busca do VisionDraughts é aprimorado, uma vez que substitui o algoritmo de busca *Minimax* pelo algoritmo Alfa-Beta combinado com TT e a estratégia de busca por aprofundamento iterativo. Este jogador, assim, como os jogadores descritos nas seções 3.2.2.3, 3.2.2.4 e 3.2.2.5 foi desenvolvido pela equipe ao qual o presente trabalho se insere.

Desta forma, o VisionDraughts apresentou desempenho muito superior ao Neuro-Draughts. Resultados apresentados em [51] comprovam que o VisionDraughts é, pelo menos, 50% mais eficiente no que se refere à sua atuação em disputas e 95% mais ágil em relação ao tempo de busca.

3.2.2.3 MP-Draughts

O MP-Draughts (*MultiPhase Draughts*) é um sistema multiagente para Damas composto por 26 agentes: 1 especialista em fases iniciais e intermediárias denominado *Initial / Intermediate Game Agent* (IIGA) e os demais são especialistas em fases de final de jogo (tabuleiros com no máximo 12 peças). Cada agente apresenta a mesma arquitetura geral do jogador VisionDraughts. Na dinâmica de jogo do MP-Draughts, um agente, o IIGA, inicia a partida e a conduz até que o tabuleiro tenha, no máximo, 12 peças. Neste momento, uma rede Kohonen-SOM irá definir, dentre os agentes de final de jogo, o agente mais adequado para lidar com o estado de tabuleiro corrente. Uma vez este agente sendo definido, ele conduzirá a partida até o final.

Cada agente de final do jogo do MP-Draughts é treinado para lidar com um determinado grupo de estados de tabuleiro de final de jogo diferente, os quais são agrupados seguindo um critério mínimo de similaridade. Ao todo são 25 grupos de treinamento que foram minerados de uma base de dados com cerca de 4000 estados de tabuleiros através de uma rede Kohonen-SOM [54]. Esta rede também é responsável por selecionar o agente de final de jogo que deve substituir o IIGA na fase de final de jogo.

A proposta deste jogador teve como objetivo diminuir a quantidade de *loops* de final de jogo que ocorriam em partidas envolvendo seus predecessores, NeuroDraughts e VisionDraughts, e prejudicavam a finalização dos jogos. Um *loop* ocorre quando o agente, mesmo em vantagem, não consegue pressionar o adversário e alcançar a vitória. Ao invés disso, o agente começa uma sequência repetitiva de movimentos (*loop*) alternando-se

entre posições inúteis do tabuleiro, sendo que esses movimentos não modificam o estado do jogo. Desse modo, a visão especializada do multiagente contribui para que o número de *loops* sejam minimizados (ou extintos), uma vez que as habilidades de final de jogo de um agente de final de jogo é substancialmente melhor que a visão de um monoagente.

Os resultados obtidos pelo MP-Draughts indicaram que, apesar do aumento no tempo de treino por conta do maior número de agentes a serem treinados (o MP-Draughts gastou 86% a mais do tempo requerido pelo VisionDraughts), o uso de uma arquitetura multiagente aumentou o desempenho do jogador. Em um torneio de 20 jogos contra o VisionDraughts, o MP-Draughts obteve 65% de vitórias [29].

3.2.2.4 D-VisionDraughts

O D-VisionDraughts (*Distributed VisionDraugths*) [60] é um sistema monoagente jogador de Damas que altera o módulo de busca sequencial Alfa-Beta implementado no VisionDraughts pelo algoritmo de busca distribuído YBWC [33] e faz uso de heurísticas para a ordenação da árvore de jogo. Esta versão contou com 10 processadores no processo de distribuição da busca. Resultados experimentais mostraram que o tempo de execução da busca do D-VisionDraughts é cerca de 83% inferior à requerida pelo VisionDraughts. Além disso, o D-VisionDraughts obteve 15% a mais de vitórias que seu oponente. Ressalta-se que esta foi a primeira versão do D-VisionDraughts, que como o VisionDraughts, implementou 12 características para representar o tabuleiro através do mapeamento NET-FEATUREMAP.

Em Ref. [19] foi proposta uma segunda versão do D-VisionDraughts. Nela foi analisado os resultados obtidos quando o número de características para representar o tabuleiro varia entre 14 e 16 e o número de processadores varia entre 10 e 16. Os resultados mostraram que, considerando as mesmas características no mapeamento NET-FEATUREMAP e mesma profundidade de busca para ambas as versões do D-VisionDraughts, a segunda versão do jogador, operando com 16 processadores, apresentou um tempo de treinamento 43% menor do que o gasto pela primeira versão. Além disso, em torneio realizado entre estas duas versões, a segunda versão do D-VisionDraughts (com 14 características no mapeamento NET-FEATUREMAP) apresentou desempenho superior de 10% em relação à primeira versão.

3.2.2.5 D-MA-Draughts

O D-MA-Draughts (*Distributed Multi-agent Draughts*) [3] estende e refina duas das arquiteturas jogadoras bem sucedidas apresentadas anteriormente, que são: MP-Draughts (seção 3.2.2.3) e D-VisionDraughts (seção 3.2.2.4). Apesar do bom desempenho apresentado por ambos jogadores, eles ainda apresentam pontos que podem ser melhorados. Desta forma, o D-MA-Draughts foi concebido de tal forma a aproveitar as melhores qualidades dos jogadores MP-Draughts e D-VisionDraughts.

Basicamente, o jogador D-MA-Draughts [3], assim como o MP-Draughts, corresponde a um SMA composto por 26 agentes sendo o primeiro IIGA e os demais especializados em fases de final de jogo. Cada um destes agentes consiste de uma rede MLP treinada sem supervisão humana por meio dos métodos TD(λ). O melhor movimento, assim como o D-VisionDraughts, é indicado pelo algoritmo YBWC, que pode ser executado através de duas estratégias de busca distintas: profundidade limitada ou por aprofundamento iterativo. Cada agente de final de jogo é qualificado a indicar movimentos (ações) para um determinado perfil de tabuleiro de final de jogo. Estes perfis são definidos por um processo de agrupamento (clusterização) realizado através de uma rede Kohonen-SOM a partir de uma base composta por estados de tabuleiro de final de jogo obtidos em partidas reais. Além do processo de agrupamento de estados de tabuleiros, a rede Kohonen-SOM também é utilizada pelo D-MA-Draughts para indicação do agente que irá conduzir a partida em fases de final de jogo. Nos jogos de competição, o D-MA-Draughts pode atuar segundo duas dinâmicas no que diz respeito à iteração entre os agentes: a primeira dinâmica envolve menor cooperação entre eles, ao passo que na segunda há um nível maior de cooperação.

Todas as melhorias obtidas pelo D-MA-Draughts foram comprovadas em cenários de testes apresentados em Ref. [3] realizados em relação a seus predecessores MP-Draughts e D-VisionDraughts. Em suma, ao verificar o *look-ahead* atingido pelos SMAs foi verificado que o D-MA-Draughts conseguiu alcançar até 4 níveis mais profundos que o MP-Draughts. Em relação ao tempo de treinamento dos SMAs, o tempo gasto pelo D-MA-Draughts foi 28,46% menor do que o tempo exigido pelo MP-Draughts. Além disso, nos torneios realizados entre o D-MA-Draughts e seus predecessores D-VisionDraughts e MP-Draughts, a proposta multiagente atuante em ambiente de alto desempenho do D-MA-Draughts mostrou-se superior em todos os jogos considerando ambas dinâmicas de atuação em partidas.

Um ponto a se destacar é que apesar das duas dinâmicas do D-MA-Draughts terem apresentado resultados positivos ao atuarem em partidas, não foi concluído qual dinâmica de fato é a mais adequada para representar o jogador. Por esta razão, um dos objetivos deste trabalho é realizar esta análise de modo a definir a dinâmica que o D-MA-Draughts passará a adotar nas partidas em que participar. Adicionalmente, salienta-se que o este jogador será utilizado como base para o cumprimento dos objetivos deste trabalho. Maiores detalhes deste jogador serão apresentados no Capítulo 7.

3.2.2.6 Outros Jogadores Automáticos de Damas Não Supervisionados

Fogel e Chellapilla em [15] utilizaram um processo co-evolutivo para implementar um jogador automático de Damas que fosse capaz de aprender a jogar sem utilizar conhecimento humano na forma de características específicas do domínio do jogo. O melhor jogador obtido neste trabalho, denominado Anaconda, foi resultado da evolução de 30

MLPs ao longo de 840 gerações. Cada geração teve em torno de 150 jogos de treinamento. Foram necessários 126.000 jogos de treinamento e 6 meses de execução para obter o Anaconda. Em um torneio de 10 jogos contra uma versão de baixo desempenho do Chinook, o Anaconda obteve 2 vitórias, 4 empates e 4 derrotas. Esse resultado foi suficiente para classificá-lo como *expert* [67].

Um jogador não supervisionado, recentemente proposto, é o de Cheheltani e Ebadzadeh [68]. Este jogador é um agente imunológico difuso que usa células de memória (*memory cells*) para representar as ações (movimentos) do jogo de Damas. Estas células ajudam um sistema de inferência difuso de Mamdani (*Mamdani Fuzzy Inference Engine* (FIS)) a decidir qual é o melhor movimento a ser executado. Para tanto, são considerados os estados anteriores e posteriores ao estado corrente do jogo. Segundo o autor, num torneio de 50 jogos contra o jogador de Fogel, o agente imunológico obteve 66% de vitórias contra 10% de vitórias do oponente.

Outro agente foi proposto recentemente por Al-Khateeb e Kendall [69], [70]. Ele corresponde a uma evolução do jogador de Fogel, o qual acrescenta um mecanismo de *aprendizagem social* e *individual* à fase de aprendizagem da MLP co-evolutiva. De acordo com os autores, este jogador é superior à versão de Fogel em [71].

Capítulo 4

Comparação Conceitual Entre os Algoritmos YBWC e APHID

Este capítulo apresenta um estudo comparativo conceitual entre as abordagens dos algoritmos YBWC e APHID, cumprindo assim, o objetivo específico 2 deste trabalho descrito na seção 1.3.1.

Conforme apresentado na seção 2.4.2, a tarefa de distribuir o algoritmo de busca Alfa-Beta não é trivial, uma vez que este algoritmo tem uma natureza serial e necessita das informações providas pelos nós localizados mais a esquerda da árvore de busca estreitar os limites de sua janela de busca e aumentar o número de podas. No entanto, nas versões distribuídas nem sempre estas informações estão disponíveis no momento oportuno, fato que resulta em sobrecarga de busca. Além disso, as versões distribuídas precisam lidar com a sobrecarga de comunicação, visto que os dados tem que ser repassados para os processadores envolvidos. Neste sentido, diversas versões de distribuição do Alfa-Beta tem sido propostas na literatura [1, 12, 13, 30, 31, 32, 33] baseadas em duas abordagens de paralelismo, síncrona e assíncrona, com o objetivo de obter um equilíbrio entre as fontes de ineficiência e o desempenho oferecido pelo algoritmo para que seja possível explorar níveis mais profundos da árvore de busca em problemas de alta complexidade. As principais versões distribuídas propostas do Alfa-Beta foram abordadas no capítulo 3 deste trabalho, onde, dentre elas, destacam-se o síncrono YBWC [33] e o assíncrono APHID [1].

De fato, cada abordagem de distribuição possui pontos fortes e limitações. Portanto, este estudo visa nortear o desenvolvimento da proposta do algoritmo distribuído contido nos objetivos deste trabalho. Assim sendo, este capítulo está organizado como se segue: a seção 4.1 apresenta como é a comunicação entre os processadores; a seção 4.2 mostra como os algoritmos trabalham para a sincronização dos resultados das tarefas repartidas entre os processadores; a seção 4.3 aborda como os algoritmos trabalham com a janela de busca Alfa-Beta; a seção 4.4 mostra como os algoritmos operam com a utilização de TT em arquitetura de memória distribuída; 4.5 expõe como é tratada a questão da portabilidade

dos algoritmos de modo a identificar a dificuldade de inclui-los em um programa que já utiliza uma versão serial do Alfa-Beta; e, por fim, a seção 4.6 apresenta as considerações finais.

4.1 Comunicação entre Processadores

Comunicação entre processadores é uma característica básica dos algoritmos distribuídos, uma vez que deve haver um canal que possibilite o envio e o recebimento de informações para que cada processador tenha condições de se direcionar corretamente para o conjunto de instruções que deve executar. A forma mais comum para realizar esta comunicação é por troca de mensagens. Neste contexto, tanto o YBWC quanto o APHID possui o seu próprio conjunto de mensagens que são trocadas entre os processadores a fim de permitir a evolução da busca.

Devido ao sincronismo e ao modo de funcionamento do YBWC (detalhado na seção 3.1.1.3), a troca de mensagens apresenta um volume muito elevado, uma vez que são identificadas as seguintes mensagens:

Início da Busca: notificação do processador principal aos demais informando o início da busca, para que estes iniciem suas requisições de tarefa.

Fim da Busca: notificação do processador principal aos demais informando que a solução final da busca foi encontrada e que eles podem cessar as requisições de tarefas.

Requisição de tarefa: mensagem enviada por um processador A a um processador B requisitando uma tarefa.

Resposta a Requisição de Tarefa - A: Mensagem enviada por um processador que recebeu uma requisição de tarefa com retorno positivo (a mensagem contem a tarefa compartilhada).

Resposta a Requisição de Tarefa - B: mensagem enviada por um processador que recebeu uma requisição de tarefa informando que não há tarefas disponíveis.

Resultado de Tarefa: retorno da avaliação da tarefa explorada pelo processador em questão.

Nova Janela de Busca: informa os novos limites da janela de busca identificados por algum processador durante o processo de busca.

Poda: mensagem enviada a algum processador escravo quando é identificado que a tarefa de sua responsabilidade é irrelevante à solução final da busca, desta forma, ela deve ser interrompida imediatamente.

O APHID foi construído de modo que há menos tipos de mensagens trafegando na rede. Tal fato está relacionado à sua abordagem assíncrona que concentra a comunicação a uma estrutura: a AphidTable, seção 3.1.2.2. Neste contexto, o APHID trabalha com as seguintes mensagens:

Envio de Tarefa: o processador mestre grava as tarefas na estrutura AphidTable de cada processador escravo.

Valor hipotético: o processador mestre envia o valor mais recente obtido da exploração de sua subárvore para formação da janela de busca nos escravos.

Resposta de Tarefa: os processadores escravos gravam na sua porção da estrutura AphidTable do mestre o resultado de suas explorações.

Finalização da Busca: mensagem enviada do mestre a todos os escravos informando que a busca principal finalizou. É importante ressaltar que esta é a única mensagem em que não é utilizada a AphidTable como intermediária de comunicação.

4.2 Sincronizações dos Resultados das Tarefas Distribuídas

Para que os algoritmos paralelos cheguem à solução ao qual foram propostos, é imprescindível que eles obtenham as informações que foram distribuídas entre o conjunto de processadores disponíveis e finalizem as instruções que irão conduzi-los à solução final. Para isso, deve existir uma sincronização dos resultados que foram explorados separadamente. Neste contexto, cada abordagem de paralelização propõe um modelo de sincronização dos dados de modo a permitir que o algoritmo centralize os resultados dos nós explorados por outros processadores e retorne a solução final.

A Figura 30 ilustra onde está localizado os pontos de unificação (sincronização) de tarefas no YBWC e no APHID.

Conforme é possível observar, o YBWC considera diversos pontos de sincronização, ou seja, um processador P_i que está explorando um nó n_x, e que possui tarefas refentes a n_x distribuídas entre os processadores P_j e P_k, apenas irá finalizar a exploração de n_x quando P_j e P_k enviarem suas soluções. Este fato pode fazer com que P_i fique ocioso determinados períodos.

O APHID tenta eliminar pontos de sincronização durante a execução do algoritmo a fim de não haver períodos de ociosidade entre os escravos. Para isso, ele estabelece um canal de comunicação a partir da estrutura AphidTable. Durante toda a busca, quando um processador escravo P_j desejar enviar uma solução de alguma tarefa para o mestre P_i, ele irá gravar tal informação na AphidTable. P_i por sua vez, quando necessitar de

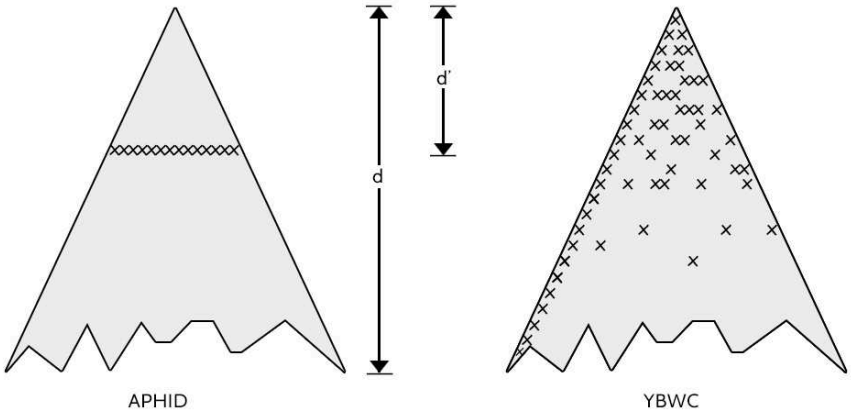


Figura 30 – Pontos de sincronização em uma busca APHID e YBWC [1].

informações de algum processador escravo P_j, irá procurar por elas na AphidTable. Caso alguma informação não esteja disponível, significa que P_j ainda não finalizou a execução de tal tarefa. No entanto, P_i não fica "travado" aguardando uma tarefa, ele continua seu processo, pois em algum momento esta informação estará disponível. Neste caso, a busca apenas finalizá quando P_i constatar que possui todas as informações de que necessita para concluir a busca principal. No APHID, cada processador escravo irá executar suas atividades de modo independente, logo não haverá outro meio de sincronização de dados dentro do algoritmo a não ser o estabelecido pela AphidTable. Salienta-se que o mestre atualiza sua AphidTable ao receber uma mensagem do tipo "Resposta de Tarefa" - descrita na seção **sc:comunicacao!** (**sc:comunicacao!**) - por meio de uma *thread* dedicada a lidar com esta operação de atualização. Neste sentido, é importante observar que diversos processadores escravos podem enviar este tipo de mensagem simultaneamente.

4.3 Janela de Busca Alfa-Beta

No algoritmo Alfa-Beta, todo estreitamento de janela produzido pelo processamento de qualquer ramo ocasiona uma imediata atualização da janela que será utilizada no processamento do próximo ramo. Isto é feito a fim de melhorar a eficiência do algoritmo através da poda. Neste sentido, as abordagens paralelas para o Alfa-Beta devem propor meios de realizar o compartilhamento de informações dos limites da janela de busca, de modo a reduzir a sobrecarga da busca. Particularmente, os algoritmos YBWC e APHID tem mecanismos muito distintos para tratar a janela de busca.

No YBWC, o algoritmo tenta manter os processadores envolvidos na busca com as informações mais atualizadas em relação à definição de janela de busca. Isto permite que o algoritmo se comporte de maneira mais próxima ao algoritmo serial no que tange à atualização da janela de busca, fato que visa minimizar a sobrecarga de busca. A Figura 31 apresenta um exemplo do mecanismo de compartilhamento das informações da janela

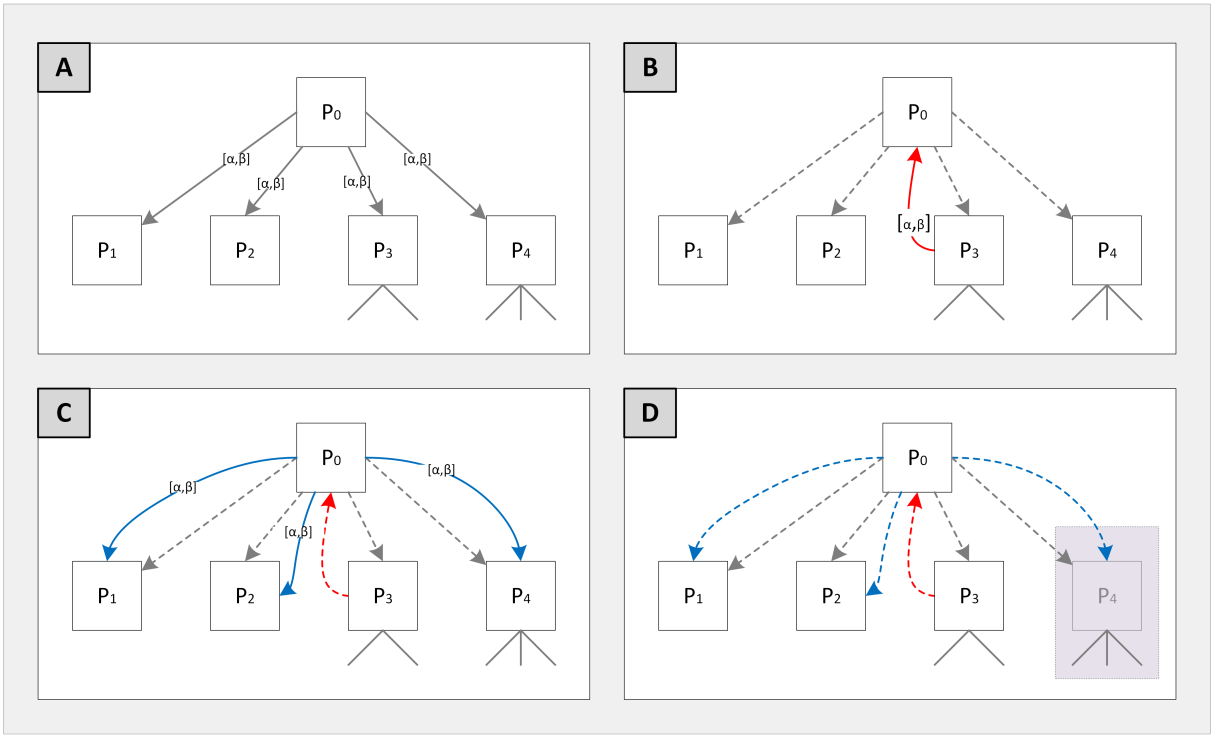


Figura 31 – Mecanismo de atualização da janela alfa-beta no algoritmo YBWC.

alfa-beta no YBWC. Neste exemplo, o processador P_3 identifica um estreitamento de janela de busca e repassa esta informação ao seu mestre (P_0). P_0, por sua vez, informa aos demais processadores sobre este estreitamento. Assim, caso algum processador identifique que a tarefa que estava explorando é inútil à solução final do algoritmo (como é o caso apresentado na Figura 31(D)), ele a interrompe. Apesar de o YBWC tentar manter a exploração da árvore de busca o mais próximo possível da versão serial, pode ocorrer de a informação de estreitamente da janela de busca não ser obtida no tempo ideal. Neste caso, é inevitável que tarefas sejam exploradas desnecessariamente contribuindo para a sobrecarga de busca.

No APHID, não há esta atualização de janela entre os processadores escravos. Cada processador irá trabalhar com sua própria janela de busca. Nesta direção, a definição desta janela segue uma política que considera o valor da varredura atual do mestre por sua subárvore, o chamado valor hipotético. Assim, a largura da janela estará condicionada a uma constante definida nas configurações do algoritmo e será calculada por meio das equações 6 e 7 apresentadas na seção 3.1.2.2. A Figura 32 ilustra o mecanismo de atualização da janela alfa-beta no APHID. Quando P_0 gera o valor hipotético, ele grava tal informação na sua AphidTable (com cópia aos escravos). Os escravos (P_1, P_2, P_3 e P_4), por sua vez, devem ficar atentos a sua AphidTable local, de modo a verificar as atualizações deste valor hipotético. É importante destacar que o valor da constante considerada na formação da janela de busca é definido de modo empírico e manual. Tal fato

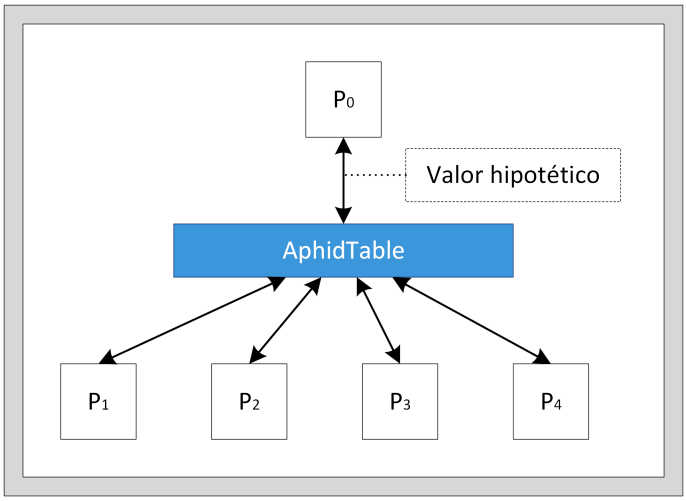


Figura 32 – Mecanismo de atualização da janela alfa-beta no algoritmo APHID.

apresenta uma fragilidade ao algoritmo, pois o valor desta constante pode comprometer fortemente a performance do algoritmo gerando algum efeito colateral indesejável: se a largura da janela resultante da aplicação desta constante ao valor hipotético for muito pequena, o algoritmo se tornará mais rápido, todavia, ele poderá perder a melhor solução da busca, pois ela poderá ficar fora dos limites desta janela. Neste caso, a resposta final do algoritmo pode ser diferente da resposta que seria retornada pela versão serial do Alfa-Beta. Por outro lado, se esta "largura" for muito grande, o algoritmo ficará mais lento, e, consequentemente, aumentará a sobrecarga de busca, uma vez que diversos nós desnecessários serão explorados. Conforme pode ser observado, a política de formação ou atualização de janela do APHID pode fazer com que o algoritmo retorne uma solução diferente daquela que a versão retornaria para um mesmo estado e profundidade de busca.

4.4 TT em Arquitetura de Memória Distribuída

Uma técnica que auxilia na performance de algoritmos de busca para jogos, em especial o jogo de Damas, é a utilização de TT. Durante o processo de busca, antes de um estado ser processado, é verificado sua existência na TT. Caso ele exista e satisfaça um conjunto de restrições que serão apresentadas na seção 5.8.1, o algoritmo pode fazer uso deste estado. Caso não exista, ele é adicionado. Neste sentido, o cenário ideal para algoritmos distribuídos é o uso de TT global (seção 2.4.3). Em situações em que não é possível trabalhar com uma TT global devido à arquitetura disponível, pode-se utilizar a TT distribuída.

O YBWC, devido a sua natureza de divisão de tarefas que funciona por roubo de trabalho (*working-stealing*), apresenta melhor performance com TT global. Por exemplo, em arquitetura de memória distribuída, um estado e_x que foi explorado por um processador P_j, pode aparecer em outro momento para o processador P_k. No entanto, como P_k não

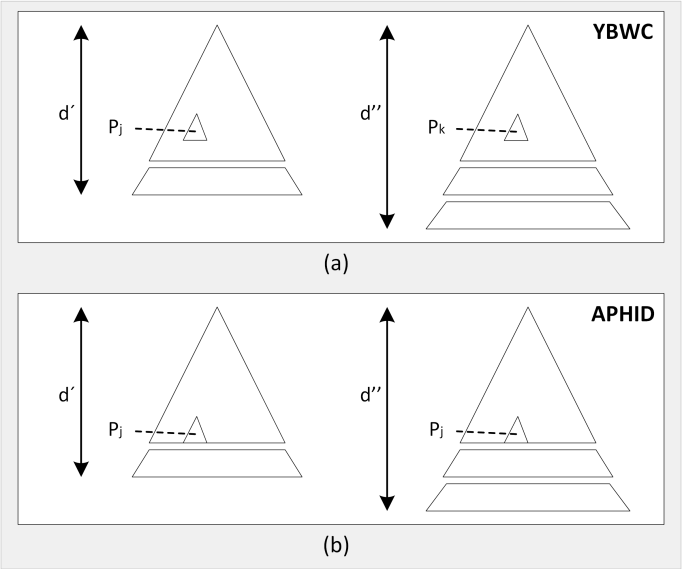


Figura 33 – Exemplificação da alocação de tarefas a um processador quando utilizada a estratégia de busca por Aprofundamento Iterativo (a) no YBWC e no APHID (b)

tem acesso às informações da TT de P_j, ele vai reprocessar e_x novamente. Esta situação é agravada quando é considerado o uso da estratégia de busca ID (seção 2.2.2), pois, considerando que o YBWC realizou a exploração de um dado estado e_x até a profundidade $d^{'}$ na iteração i, quando ele realizar a exploração de e_x novamente na profundidade $d^{''}$ na iteração $i+1$, não haverá garantias de que a distribuição das tarefas ficará sob a responsabilidade dos mesmos processadores. Este caso é ilustrado na Figura 33(a), onde na iteração $d^{'}$ um estado é explorado por P_j, no entanto, na próxima iteração na profundidade $d^{''}$ ele passa a ser explorado por P_k. Todavia, mesmo considerando estes fatores, é preferível a utilização de TT distribuída a não utilizar.

O APHID consegue operar bem com TT distribuída. Como o algoritmo irá explorar um determinado estado e_x até a profundidade $d^{'}$, e os nós folhas desta exploração serão distribuídos entre os processadores escravos, o algoritmo tenta manter, o máximo possível, a atribuição dos nós aos mesmos processadores durante toda a busca. A Figura 33(b) ilustra esta situação, onde um estado e_x que foi atribuído ao processador P_j na iteração $d^{'}$ será mantido com P_j na iteração $d^{''}$, e assim sucessivamente até o final da busca. e_x apenas será atribuído a outro processador se houver a necessidade de realocação de atividades devido a uma sobrecarga de P_j (atividade de balanceamento de carga). Isto significa que, uma vez atribuída uma tarefa a um processador, esta tarefa tem a chance de permanecer com tal processador até o final. Neste caso, a estratégia de busca por aprofundamento iterativo não causa impacto negativo na performance do algoritmo. Além disso, o APHID também implementa um mecanismo cíclico de compartilhamento das informações de TT (conforme detalhado na seção 3.1.2.2), fato que aumenta a possibilidade do algoritmo usar informações globais de nós já avaliados.

Ressalta-se que neste trabalho, todos os experimentos foram executados em arquitetura de memória distribuída. Além disso, a proposta deste trabalho de doutorado em desenvolver uma nova versão paralela do algoritmo Alfa-Beta também foca em arquitetura de memória distribuída, o que é um desafio para qualquer abordagem de paralelização do Alfa-Beta.

4.5 Portabilidade

Um dos fatores que influencia na escolha de um algoritmo a ser inserido em um problema é a complexidade da implementação sobre a aplicação. Se esta complexidade for pequena, o desenvolvedor e/ou pesquisador terá maior motivação em adotar tal algoritmo.

Para lidar com esta situação, a ideia do autor do APHID em [1] foi desenvolver uma biblioteca independente. Desta forma, sua integração com aplicações já existentes, que utilizam a versão do Alfa-Beta serial, se torna simples e sem muito impacto para o programador: através de algumas configurações sem complexidade é possível ter a versão distribuída do Alfa-Beta funcional. De fato, a biblioteca do APHID foi disponibilizada durante alguns anos na WEB, todavia, na execução deste trabalho, tal biblioteca não estava disponível. No entanto, a ideia de disponibilizar o algoritmo através de uma biblioteca continua sendo muito interessante, pois cria um meio de facilitação permitindo que outros pesquisadores tenham interesse em aplicar o algoritmo a problemas cujo o propósito seja satisfeito. Neste sentido, tais pesquisadores podem fornecer melhorias ou até mesmo evoluir o algoritmo de busca.

O YBWC proposto por [33] não oferece esta facilidade. Para inserir este algoritmo em uma aplicação que já utiliza a versão serial do Alfa-Beta, o programador terá que inserir o mecanismo de troca de mensagens no meio de seu programa. Esta tarefa demanda conhecimento do programador em relação ao processo de trocas de mensagens. Além disso, o processo como um todo se torna ligeiramente complexo.

4.6 Considerações finais

Este capítulo apresentou uma comparação conceitual das abordagens síncrona do algoritmo YBWC e assíncrona do APHID. Os tópicos estudados foram a comunicação entre os processores, a sincronização dos resultados das tarefas distribuídas, a janela de busca alfa-beta, o uso de TT em arquitetura de memória distribuída e a portabilidade dos algoritmos para serem inseridos em outras aplicações. Assim, foi possível observar que a versão do APHID parece muito apropriada para arquitetura de memória distribuída devido à forma como opera com a TT e por não demandar trocas de mensagens em relação ao estreitamento dos limites da janela de busca. Todavia, o APHID dispõe de uma fragilidade representada pela política de formação e atualização da janela de busca

nos processadores escravos. De fato, esta política pode afetar, inclusive, a qualidade da solução retornada, visto que permite a geração de uma janela de busca muito estreita que poderá perder, com frequência, a solução ótima. Por outro lado, o YBWC não apresenta este tipo de limitação, pois ele tenta obter a janela de busca mais atual e compartilhar as informações de poda e estreitamento da janela de busca entre todos os processadores. No entanto, esta estratégia pode ocasionar períodos de ociosidade devido às sincronizações dos dados. Além disso, tal versão, conceitualmente, apresenta melhores resultados em arquitetura de memória compartilhada, pois neste contexto haveria maiores benefícios a utilização de TT.

A fim de complementar este estudo, o presente trabalho contemplou uma comparação prática entre os algoritmos YBWC e APHID aplicados a um mesmo problema de alta complexidade. Para isso, foi adotado o domínio do jogo de Damas. Neste sentido, foi utilizado um agente jogador de Damas automático já implementado pela autora deste trabalho, o D-VisionDraughts [19], que utiliza o YBWC como mecanismo de tomada de decisão - detalhes deste jogador podem ser visualizados na seção 3.2.2.4. No caso do APHID, foi necessário implementa-lo no âmbito deste trabalho[1] com base na proposta do autor do APHID [1], Brockington. Adicionalmente, foi construído um agente automático jogador de Damas com arquitetura análoga à do D-VisionDraughts, mas com as tomadas de decisão conduzidas pelo algoritmo APHID. Por esta razão este agente foi denominado APHID-Draughts [22].

É importante ressaltar que em [1], Brockington realizou algumas comparações técnicas entre as abordagens síncronas (incluindo o YBWC) e o algoritmo APHID. Estas comparações focaram apenas em parâmetros relacionados às fontes de ineficiência (seção 2.4.2) e velocidade de execução do algoritmo. Os experimentos realizados foram conduzidos em arquitetura de memória compartilhada. No caso particular da comparação entre o YBWC e o APHID, o autor utilizou o domínio do jogo Othello através do jogador Keyano[23] como estudo de caso, no entanto, foram avaliados somente movimentos individuais a fim de focar nos parâmetros inerentes à avaliação proposta. De fato, não existe na literatura uma comparação entre os algoritmos APHID e YBWC aplicados a um mesmo problema prático. O próprio Brockington sugeriu em [1] que uma comparação desta natureza seria muito apropriada para colocar em perspectiva o desempenho real destes algoritmos em uma situação prática, pois seria possível avaliar a eficiência de cada algoritmo em relação às suas tomadas de decisões para atingir o objetivo esperado.

Os experimentos realizados e a análise dos resultados obtidos desta comparação prática serão apresentados no Capítulo 8 que relaciona todos os experimentos desenvolvidos neste trabalho.

[1] Em [1] o autor disponibilizou o algoritmo APHID como uma biblioteca independente na WEB. Todavia, tal biblioteca não está mais disponível e, apesar da autora desta qualificação ter entrado com contado com o autor do algoritmo, o mesmo não enviou a referida biblioteca. Desta forma, o APHID foi integralmente implementado neste trabalho.

Capítulo 5

ADABA: Uma nova versão distribuída do Alfa-Beta com abordagem assíncrona de paralelismo

Este capítulo se refere ao objetivo específico 3 traçado neste trabalho e detalhado na seção 1.3.1. Nesta direção, apresenta-se o *Aysnchronous Distributed Alpha-Beta Algorithm* (ADABA): uma nova versão distribuída do algoritmo de busca Alfa-Beta que segue o modelo mestre-escravo de paralelismo segundo a abordagem assíncrona. O ADABA é inspirado nas ideias de distribuição do algoritmo APHID (seção 3.1.2.2). A escolha do APHID foi motivada pelos seguintes fatores:

- Os estudos realizados em relação à sua dinâmica de operação apresentados no Capítulo 4 indicaram que a abordagem de paralelismo assíncrona deste algoritmo mostra-se adequada tanto para arquiteturas de memória compartilhada quanto de memória distribuída. Este último tipo de arquitetura tem a vantagem de ser mais acessível economicamente para infraestruturas que não contam com os hardwares necessários para a montagem de uma arquitetura de memória compartilhada.

- A proposta do APHID apresentou bons resultados na literatura [1]. Além disso, os experimentos realizados pela autora do presente trabalho em [21] e apresentados no Capítulo 8 em uma análise prática que compara o desempenho dos algoritmos síncrono YBWC e assíncrono APHID sobre o mesmo problema de alta complexidade - a saber, o domínio do jogo de Damas -, mostraram que o APHID se sobressaiu significativamente em relação ao YBWC.

Apesar das excelentes contribuições associadas ao APHID, nos estudos relacionados a este algoritmo realizados no cumprimento do objetivo específico 1 deste trabalho (seção 1.3) foram identificados os seguintes pontos que podem comprometer o desempenho da busca:

- a política de definição e atualização da janela de busca nos processadores escravos depende de uma constante cujo valor é estipulado empírica e manualmente. Desta forma, tal política pode ocasionar dois inconvenientes: perda da acuidade da tomada de decisão e sobrecarga de busca.

- o mestre aloca apenas uma *thread* para gerenciar os resultados recebidos dos processadores escravos. Neste caso, o problema é que diversos escravos podem enviar simultaneamente o resultado da exploração das tarefas que lhes foram alocadas para que o mestre atualize a busca principal. Todavia, o mestre pode atender apenas um escravo de cada vez, o que compromete a celeridade do processo de busca. Além disso, o mestre utiliza estes resultados para gerar dados para definir a janela de busca que será utilizada pelos escravos, portanto, quanto mais informação disponível, mais precisos serão os dados que serão fornecidos.

- a política de seleção de tarefas de maior prioridade utilizada pelos escravos para ordenar as atividades recebidas do mestre é primordialmente baseada na quantidade de trabalho associada à tarefa em questão (quanto mais rasa for a subárvore associada a esta tarefa, mais trabalho ela representa), ao invés de priorizar a proximidade das tarefas com a solução da busca. Mais especificamente, o APHID apenas leva em consideração o critério "distância da solução" quando existem duas ou mais tarefas candidatas no mesmo nível da subárvore. Neste caso, o referido critério é utilizado para decidir a próxima tarefa a ser selecionada pelo processador escravo.

Sendo assim, o ADABA aprimora a dinâmica geral do APHID tratando estes pontos da seguinte forma:

- Propõe uma nova política automatizada para formar e atualizar a janela de busca nos processadores escravos.

- Cria um grupo de *threads* inerentes ao processo mestre para gerenciar as tarefas recebidas dos escravos.

- Utiliza uma política de prioridade para a ordenação das tarefas que os escravos vão explorar focada essencialmente na "distância da solução" dos nós considerando que eles estão ordenados do modo "melhor-para-pior" (*best-to-worst*).

Ressalta-se que as versões assíncronas, devido à suas dinâmicas de operação, podem prover uma solução distinta daquela que a versão serial retornaria para o mesmo estado e mesma profundidade de busca. Por esta razão, as propostas do ADABA foram estudadas considerando um balanceamento entre a velocidade da busca e a acurácia da solução. Esta acurácia é avaliada ao comparar os resultados retornados pelo algoritmo distribuído e aqueles obtidos pela versão serial nas mesmas circunstâncias, uma vez que esta última

sempre retorna a solução ótima. Os resultados referentes ao desempenho deste algoritmo serão apresentados no Capítulo 8.

As seções deste Capítulo estão organizadas da seguinte forma: a seção 5.1 apresenta um panorama geral do ADABA; a seção 5.2 apresenta as versões do algoritmo Alfa-Beta serial utilizadas internamente pelo ADABA; a seção 5.3 mostra a nova política de atualização da janela de busca; a seção 5.4 apresenta a nova estratégia de tratamento das tarefas recebidas dos escravos; a seção 5.5 mostra como os estados são representados para o algoritmo de busca ADABA; a seção 5.6 apresenta os detalhes da estrutura de comunicação entre mestre e escravos; a seção 5.7 mostra como o ADABA implementa o balanceamento de carga; a seção 5.8 apresenta a estrutura das TTs utilizadas no curso do algoritmo; a seção 5.9 apresenta detalhes da utilização da estratégia de ID no ADABA; e, por fim, a seção 5.10 apresenta as considerações finais.

5.1 Visão geral do ADABA

O ADABA possui uma hierarquia de apenas um nível composta por n processadores, sendo um processador, P_0, atuando no papel de mestre e os demais P_i processadores como escravos, em que $(0 < i < n)$. Toda a comunicação entre P_0 e P_i é feita por meio de uma estrutura de dados intermediária denominada *ADABA Intermediate Structure* (AIS) cujos detalhes serão apresentados na seção 5.6. É importante salientar que cada processador possui sua própria AIS, no entanto, apenas o mestre tem a visão global de todas as AISs.

P_0 é o responsável por expandir um nó raiz s até uma profundidade de busca d_m conforme ilustrado na Figura 34. Esta expansão utiliza o algoritmo de busca Alfa-Beta serial. Os nós localizados no nível d_m determinam uma borda cujos os nós existentes nela serão distribuídos entre cada P_i. Os escravos recebem as tarefas de P_0 e as processam por ID (seção 2.2.2) até uma profundidade máxima de busca d, onde $d > d_m$. Em cada iteração, os escravos registram os resultados obtidos até aquele momento na AIS de P_0. Enquanto os escravos operam, P_0 realiza diversas varreduras em sua subárvore até o nível d_m, sempre utilizando os dados mais atualizados (recebidos dos escravos) para a borda que estão registrados em sua AIS. Quando P_0 constata que todos os nós foram processados até o nível d ou que o tempo destinado para a realização do movimento foi atingido, ele notifica cada P_i para encerrar suas atividades, visto que a busca finalizou. P_0 também é responsável por retornar a solução final da busca. É importante destacar que os sucessores de um nó são ordenados do modo "melhor-para-pior" (*best-to-worst*).

A fim de detalhar as atividades desempenhadas por cada processador no ADABA, as subseções 5.1.1 e 5.1.2 apresentam, respectivamente, as instruções realizadas pelos processadores mestre e escravos.

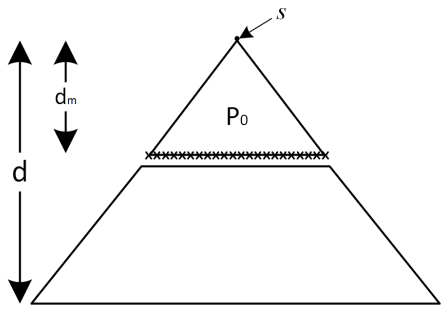


Figura 34 – Ilustração da árvore de busca de um nó s a partir do ADABA.

5.1.1 Módulo Mestre

O módulo mestre é responsável por gerenciar o processo de busca para indicar um movimento apropriado ao estado de jogo corrente s. O pseudocódigo 3 apresenta as principais instruções executadas pelo processador mestre P_0.

Pseudocódigo 3 Módulo mestre do ADABA

```
1: float masterSearch(s:Node)
2: edge:List;
3: bestmove:Move;
4: eldestBrother: Node
5: besteval, w, d_m, α_m, β_m, α_t, β_t, diff: numeric;
6: d_m ← getMasterDepth();
7: d ← getMaxSearchDepth();
8: repeat
9:    α_m ← −∞;
10:   β_m ← +∞;
11:   besteval ← alphabeta(s, d_m, α_m, β_m, bestmove, edge);
12:   eldestBrother = edge.get(0);
13:   if (n.getCurrentDepth() < d) then
14:      α_t ← −∞;
15:      β_t ← +∞;
16:      w ← alphabeta(eldestBrother, d, α_t, β_t, bestmove);
17:   else
18:      w ← eldestBrother.getEvaluation();
19:   end if
20:   diff = |besteval - w|;
21:   α_t ← α_m - diff;
22:   β_t ← β_m + diff;
23:   updateWindows();
24:   sliceBetweenSlaves(edge);
25:   sendSlavesTasks();
26:   loadBalance();
27: until searchCompleted();
28: return besteval;
```

Primeiramente, P_0 realiza uma busca, utilizando a variante do Alfa-Beta serial apresentada no pseudocódigo 6, que expande uma subárvore cuja raiz corresponde ao estado s e o limite à profundidade d_m. O valor de d_m, definido nas configurações do sistema, deve ser raso o suficiente para permitir uma varredura (passagem) rápida pela subárvore, uma vez que o mestre realizará diversas varreduras no curso da busca principal. A estrutura *edge* armazena os nós presentes no nível d_m. Em cada varredura, os valores dos limites da janela de busca iniciam como $\alpha_m = -\infty$ (linha 9) e $\beta_m = \infty$ (linha 10).

É importante ressaltar que os nós da subárvore do mestre são ordenados do modo "melhor-para-pior". Desta forma, os nós presentes em *edge* recebem uma identificação vinculada com sua posição em relação ao seu nó pai. Por exemplo, na Figura 35 os nós D, E e F são sucessores do nó B, considerando que estão ordenados, o nó D irá receber o identificador 1, porque ele é o sucessor mais a esquerda de B, e seguindo este conceito, os nós E e F irão receber, respectivamente, os identificadores 2 e 3. Os sucessores do nó C irão receber uma identificação similar, isto é, para G o identificador é igual a 1 e assim por diante. Tal estratégia foi adotada a fim de permitir a implementação da política de prioridade destes nós, que será apresentada na seção 5.1.2.

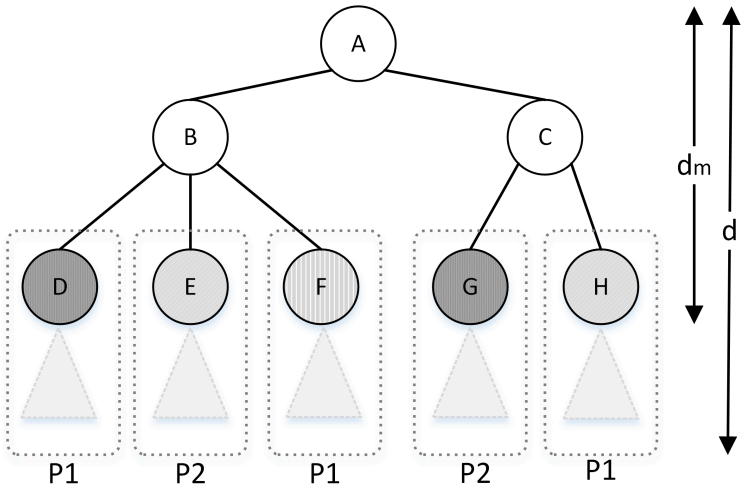


Figura 35 – Ilustração da subárvore do mestre destacando a fronteira da árvore de busca e a distribuição de tarefas entre dois processadores P_1 e P_2

Após realizar a busca Alfa-Beta no pseudocódigo 3, o mestre identifica o nó mais a esquerda da fronteira de busca e o armazena em *eldestBrother* (linha 12). Na sequência, é verificado se tal nó ainda não teve seu processamento concluído até a profundidade máxima de busca d (linha 13). Caso positivo, será realizada a exploração de *eldestBrother* até a profundidade d utilizando a janela de busca $[-\infty\text{-}\infty]$ e o valor obtido será armazenado na variável w (linha 16). Caso contrário, w receberá o valor da avaliação armazenada em *eldestBrother* (linha 18). Ressalta-se que o método *alphaBeta* executado na linha 13 não necessita receber a borda da busca como parâmetro, visto que esta é uma especificidade

da busca de varredura do mestre, portanto, ele segue as instruções descritas no pseudocódigo 5. Após a definição de w é realizado o cálculo da diferença entre o valor obtido na passagem corrente do mestre *besteval* e w (linha 20). O valor absoluto desta diferença é armazenado na variável $diff$. Além disso, ainda são calculados os valores dos limites mínimo (α_t) - linha 21 - e máximo (β_t) - linha 22 - da janela de busca que será utilizada pelos processadores escravos, segundo as expressões 8 e 9, respectivamente, que serão apresentadas na seção 5.3 sobre a política de formação da janela de busca do ADABA. Os valores dos limites α_t e β_t serão registrados na AIS dos processadores escravos a partir do método *updateWindows* (linha 23).

Os nós da borda da subárvore do mestre serão repartidos entre os escravos segundo o modo *round-robin* por meio do método *sliceBetweenSlaves* (linha 24). Na Figura 35, é possível visualizar a partição de tarefas entre dois processadores P_1 e P_2. Desta forma, as tarefas relacionadas aos nós D, F e H foram atribuídas a P_1, enquanto os nós E e G foram atribuídos ao processador P_2. Uma vez que a repartição é concluída, o mestre grava as tarefas na AIS de cada escravo através do método *sendSlavesTasks* (linha 25). O mestre constantemente verifica se há algum processador escravo sobrecarregado. Para isso, ele utiliza o método *loadBalance* (linha 26) que verifica sua AIS checando os escravos que finalizaram ou não suas atividades. Assim, aquele processador "livre" recebe tarefas de um que ainda está ocupado. Este processo é melhor detalhado na seção 5.7.

As instruções contidas entre as linhas 8 e 25 serão repetidas até que o método *searchCompleted* (linha 27) retorne o valor *true*. Isto ocorrerá quando todas as tarefas presentes na borda da subárvore do mestre tiverem sido processadas até o nível máximo d ou que o limite de tempo da busca tenha sido atingido. Uma vez finalizado, o mestre retorna a predição *besteval* associada ao estado s (linha 26).

5.1.2 Módulo Escravo

Este módulo é responsável por executar as tarefas atribuídas por P_0 e registradas na AIS local. Antes mesmo da busca principal iniciar, todos os escravos são instanciados e iniciam a execução do método *slaveSearch* apresentado no pseudocódigo 4. Cada escravo possui uma estrutura da AIS idêntica à do mestre, todavia, ela conterá apenas as informações de sua responsabilidade.

Quando uma instância de um escravo é criada, o processo fica em estado ocioso aguardando que sua AIS possua atividades (linhas 3 a 5). Ressalta-se que esta ociosidade não acarreta em danos à busca, uma vez que não há atividade alguma para ser executada pelo escravo, por exemplo, o processador mestre pode estar executando outros módulos da arquitetura do sistema em que o ADABA está inserido.

Ao ser constatado registros na AIS, o primeiro passo é escolher a tarefa t de maior prioridade em relação às demais (linha 6). Para isso, o ADABA implementa uma nova política em que as tarefas a serem processadas são essencialmente ordenadas em relação

Pseudocódigo 4 Módulo escravo do ADABA

```
1: void slaveSearch()
2: alpha, beta, current_depth: numeric;
3: while AIS is empty do
4:    wait()
5: end while
6: t ← getPrioritizedTask();
7: repeat
8:    if t is not null then
9:       current_depth ← t.getDepth() + I;
10:      alpha ← getMasterAlfa();
11:      beta ← getMasterBeta();
12:      alphabeta(t, current_depth, α, β);
13:      updateAIS(t);
14:      batch.add(t);
15:      if batch.size == maxBatchSize then
16:         sendBatch();
17:      end if
18:   end if
19: until searchCompleted();
20: slaveSearch();
```

ao critério "distância da solução", uma vez que são priorizadas aquelas que estão mais próximas da solução desejada. Assim, o algoritmo examina o identificador atribuído pelo processador mestre às tarefas, que está relacionado à localização do nó em relação ao seu "pai" na subárvore do mestre, conforme mostrado na seção 5.1.1. A ideia é tentar explorar primeiro as tarefas que podem ajudar a busca rasa realizada pelo mestre a encontrar valores que aumentam o número de podas e que possam atenuar a sobrecarga de busca (seção 2.4.2). Caso duas tarefas possuam a mesma prioridade, o nível mais raso é considerado. O método da linha 6 pode retornar um valor nulo caso não exista tarefa para ser processada. Tal fato pode ocorrer caso todos os nós existentes na AIS local tenham o status igual a *CERTAIN*, ou seja, foram explorados na profundidade máxima da busca d. Os possíveis valores de status que uma tarefa pode possuir é detalhado na seção 5.5.

Uma vez que t é selecionada, estipula-se a profundidade *current_depth* que a busca será executada (linha 9), visto que os escravos exploram suas tarefas por ID. Para isso, é considerado o valor da última profundidade em que o nó foi explorado acrescendo um valor I - correspondente a uma constante definida nas configurações do sistema - que determina o incremento do nível de busca. Por exemplo, se for a primeira expansão de t, *current_depth* será igual a $d_m + I$. Tal fato ocorre porque os resultados destas buscas mais rasas são constantemente enviados ao mestre de tal forma a lhe permitir atualizar os valores da borda de sua subárvore. Considerando que cada escravo tem sua própria TT, a exploração por ID não compromete o desempenho da busca.

Na sequência, é realizada a leitura dos limites da janela alfa-beta (linhas 10 e 11)

enviados pelo mestre. Assim, a busca Alfa-Beta serial (linha 12) é executada segundo o método apresentado no pseudocódigo 5. É importante ressaltar que caso ocorra alguma atualização dos dados da janela de busca dos escravos, eles serão considerados apenas para a próxima tarefa selecionada para exploração.

Tão logo o processamento de t seja finalizado, esta tarefa será atualizada na AIS (linha 13) e adicionada ao pacote (linha 14) que será enviado ao mestre (linhas 15-16). Tal procedimento é adotado a fim de não sobrecarregar a rede em arquiteturas de memória distribuída com o envio de diversas tarefas sequencialmente por vários processadores.

A repetição das instruções compreendidas entre as linhas 7 e 19 será realizada até que o método *searchCompleted* (linha 19) retorne o valor *true*. Isto irá ocorrer quando o escravo receber do mestre uma mensagem para finalizar a busca. Desta forma, os escravos voltam a aguardar por novas tarefas (linhas 20).

5.2 Algoritmo Alfa-beta serial no ADABA

Internamente, cada processador do ADABA irá executar a busca Alfa-Beta serial para os nós de sua responsabilidade. A versão adotada neste trabalho é a *fail-soft*, visto que é retornado o valor real da predição do estado avaliado e não o valor do limite (MIN ou MAX) que ocasionou a poda na busca, conforme ocorre na outra versão do algoritmo *fail-hard* (seção 2.3.2.1). Devido a esta característica, a versão *fail-soft* torna possível a utilização de TT. O pseudocódigo 5 apresenta a versão *fail-soft* do Alfa-Beta.

O método *alphaBeta* recebe como parâmetros o nó (*Node*) a ser avaliado, a profundidade d em que a busca deve ocorrer, o intervalo da janela de busca delimitado pelos valores *min* (referente a α) e *max* (referente a β) e *bestmove* que irá referenciar a melhor jogada que será indicada ao estado n. O retorno deste algoritmo é o valor da predição, que representa a avaliação de n do ponto de vista do agente jogador. Trata-se de um método recursivo que apresenta duas situações como ponto de parada (linha 2):

1. o nó não possui mais sucessores;

2. a profundidade máxima *depth* é atingida.

Conforme apresentado na seção 2.3, um nó pode estar em um nível minimizador ou maximizador. Caso o nó esteja em um nível de maximização (linha 5), *besteval* recebe o valor do limite inferior (*min*) da janela alfa-beta (no início da busca, este valor é configurado com o maior valor negativo possível, por exemplo, - ∞). Para cada sucessor de n (linha 7) o método *alphaBeta* é chamado com profundidade *d - 1* e com uma janela de busca constituída por $[besteval, max]$. Tal fato significa que, no nível de maximização, sempre que ocorrer atualização de um dos limites do intervalo de busca, ela será um incremento no limite inferior do seu sucessor (*child*). Desta forma, ao final do laço da

Pseudocódigo 5 Alfa-Beta versão *fail-soft*

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

linha 7, *besteval* deverá ter o valor da maior predição dos sucessores de n. O algoritmo irá retornar o valor de *besteval* caso o seu valor seja superior (ou igual) ao do limite *max* do intervalo de busca (linhas de 14 a 16).

Os nós existentes no nível de minimização vão ter operações semelhantes ao do nível de maximização, no entanto, a variável *besteval* recebe o valor do limite superior (*max*) da janela alfa-beta (linha 21). Lembrando que no início da busca este valor deve ser o menor possível, por exemplo, $-\infty$. A chamada do método *alphaBeta* utilizará uma janela configurada como $[min, besteval]$. Tal fato significa que, no nível de minimização, sempre que ocorrer atualização de um dos limites do intervalo de busca, ela será um decremento no limite superior. A variável *besteval* deverá armazenar o menor valor encontrado entre os sucessores de n (linhas de 24 a 27). Caso a predição armazenada na variável *besteval* seja menor que o limite inferior do intervalo de busca (linha 29), o algoritmo retornará o seu valor (linha 30).

Ao final da etapa tanto de maximização quanto de minimização, a variável *thebest* conterá o melhor movimento a ser executado a partir do estado n. Por esta razão, *bestmove* é atualizado com o valor de *thebest* (linhas 17 e 32). Além disso, *besteval* conterá a melhor predição de todos os filhos do estado do tabuleiro n, portanto, será retornado como valor da predição associada ao estado do tabuleiro n (linhas 18 e 33).

Variante do mestre para o Alfa-Beta versão *fail-soft*

Para que o mestre consiga realizar suas atividades de gerenciamento da busca a partir da fronteira *edge*, existe um conjunto de instruções que necessitam ser incorporadas à versão do Alfa-Beta apresentada no pseudocódigo 5. Por esta razão, pseudocódigo 6 apresenta a versão do Alfa-Beta serial utilizada pelo processador mestre.

Os pontos em que esta variante do Alfa-Beta se difere daquela apresentada no pseudocódigo 5 são:

- Além dos parâmetros comuns ao pseudocódigo 5, o algoritmo recebe a lista *edge* que armazena os nós da borda da busca do mestre (linha 1).

- Caso o algoritmo atinja a profundidade máxima enviada para o método *alphaBeta*, ele irá armazenar o nó n na lista *edge* (linhas 3 a 5).

- Como o mestre realiza inúmeras varreduras em sua subárvore de modo a atualizar seu valor *minimax*, ele deve considerar os valores mais atuais da sua fronteira retornados pelos escravos. Desta forma, primeiramente, será verificado se o nó que está sendo avaliado existe na fronteira (*edge*) do mestre, caso positivo, o valor de sua avaliação é retornado. Caso negativo, o algoritmo retorna o método *evaluate* com a predição dada pela função de avaliação para o estado do tabuleiro n (linhas 6 a 10).

Pseudocódigo 6 Alfa-Beta versão *fail-soft* variante APHID mestre

```
1: float alphaBeta(n:node,depth:int,min:int,max:int,bestmove:move, edge:List)
2: if leaf(n) or depth=0 then
3:    if depth == 0 then
4:       edge.add(n)
5:    end if
6:    if n exists in edge then
7:       return  edge.get(n).besteval
8:    else
9:       return  evaluate(n)
10:   end if
11: end if
12: if n is a max node then
13:   besteval ← min
14:   for each child of n do
15:      v ← alfaBeta(child,d-1,besteval,max,bestmove)
16:      if v > besteval then
17:         besteval ← v
18:         thebest ← bestmove
19:      end if
20:   end for
21:   if besteval >= max then
22:      return  besteval
23:   end if
24:   bestmove ← thebest
25:   return  besteval
26: end if
27: if n is a min node then
28:   besteval ← max
29:   for each child of n do
30:      v ← alfaBeta(child,d-1,min,besteval,bestmove)
31:      if v < besteval then
32:         besteval ← v
33:         thebest ← bestmove
34:      end if
35:   end for
36:   if besteval <= min then
37:      return  besteval
38:   end if
39:   bestmove ← thebest
40:   return  besteval
41: end if
```

5.3 Janela de Busca dos Escravos

A política de formação e atualização da janela de busca dos processadores escravos proposta pelo ADABA considera que cada processador irá operar com sua própria janela

de busca. A ideia é calcular limites iniciais para estas janelas de busca a fim obter o máximo das vantagens proporcionadas pelo processo de poda e também obter uma boa acurácia da solução retornada pelo algoritmo. Esta política foi inspirada nas ideias do YBWC (seção 3.1.1.3) em que há a definição de uma janela de busca inicial a partir do irmão localizado mais a esquerda da árvore de busca e, apenas após a obtenção desta janela é permitida a distribuição de tarefas. A Figura 36 ilustra esta estratégia que é detalhada na sequência.

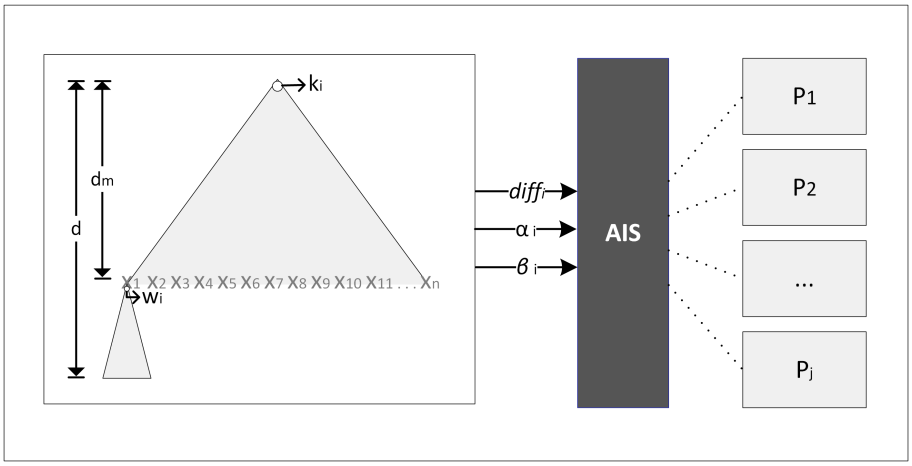


Figura 36 – Esquema de atualização das informações da janela de busca no ADABA

Em cada passagem i do mestre P_0 por sua subárvore, ele obtém os seguintes dados: o valor k_i referente à solução corrente do algoritmo (correspondente ao nó raiz), os limites α_i e β_i da janela de busca e a fronteira f_i composta pelos nós existentes no nível d_m (nós $x_1, x_2, ..., x_n$ na Figura 36). Quando a passagem i é finalizada, antes de iniciar a passagem $i+1$, o ADABA realiza o processo de calcular os valores que serão passados aos escravos, a fim de que eles definam suas janelas de busca. Para isso, P_0 seleciona o nó localizado mais a esquerda de f_i (nó x_1 na Figura 36) denominado $eldest\_brother_i$ em homenagem ao YBWC. Então, o $eldest\_brother_i$ é explorado a partir da profundidade d_m até d, que retornará um valor de avaliação referenciado como w_i. A partir deste momento, P_0 irá realizar as seguintes instruções:

- o cálculo de $diff_i$, que corresponde ao valor absoluto da diferença entre os valores de k_i e w_i;
- o cálculo para o limite inferior (α) da janela de busca a partir da expressão 8 e para o limite superior (β) a partir da expressão 9.

$$\alpha \leftarrow \alpha_i - diff_i; \tag{8}$$

$$\beta \leftarrow \beta_i + diff_i; \tag{9}$$

- a repartição dos nós da fronteira f_i, exceto o nó $eldest–brother_i$, pois ele já foi processado até a profundidade d. Na Figura 36 os nós que serão repartidos são $x_1, x_2, ..., x_n$;

- o registro na AIS de cada escravo do valor dos limites α_i e β_i obtidos pelo mestre na sua passagem i e das tarefas que foram atribuídas ao escravo em questão.

Os escravos irão utilizar os valores enviados por P_0 para definir suas janelas de busca. Desta forma, antes de iniciar a exploração de uma nova tarefa, um escravo P_i irá realizar a leitura dos valores de α_i e β_i em sua AIS para utilizar no processo de busca que segue o busca Alfa-Beta serial, conforme pseudocódigo apresentado na seção 5.2.

Caso P_i já tenha iniciado o processo de exploração de uma tarefa t_x e ocorra uma alteração nos valores de α_i e β_i resultante de uma passagem realizada por P_0, estes novos valores não irão ser considerados durante a exploração atual de t_x. Portanto, os efeitos desta alteração serão apenas aplicados quando P_i iniciar a exploração de uma nova tarefa.

Um ponto a se destacar é que como a definição dos valores de formação da janela estão vinculados à passagem i do mestre por sua subárvore, a medida que novas passagens ocorrem, a fronteira de busca pode sofrer alterações resultando em um novo nó na posição de $eldest\_brother_i$. Neste caso, é necessário considerar que:

- o nó que ocupava a posição de $eldest\_brother_i$ já concluiu a exploração no nível d, portanto, ele não será enviado aos escravos;

- o novo nó que passou a ocupar a posição de $eldest\_brother_i$, caso não esteja concluído (processado até o nível d), será explorado de d_m até d. Caso contrário, o valor de sua avaliação será utilizado para o cálculo de $diff_i$.

5.3.1 Discussão da estratégia do ADABA

De acordo com a estratégia apresentada, apenas após P_0 definir um valor para a formação ou atualização da janela de busca inicial dos escravos, ele realiza a repartição e o registro das informações necessárias para o processamento das tarefas que designou a cada P_i. Tal estratégia foi adotada, pois assim como no YBWC, o ADABA parte do princípio de que os nós de f_i estão ordenados do modo "melhor-para-pior", portanto, o nó localizado mais a esquerda tem maiores chances de prover um valor que esteja próximo à solução ótima do algoritmo, isto é, aquela que seria retornada pela versão serial.

Considerando a estratégia do ADABA, poderia, logicamente, ser levantada a seguinte questão: ao invés de ajustar os limites da janela de acordo com as expressões (8) and (9), não seria mais apropriado considerar os novos limites como aqueles que foram produzidos

ao final da exploração do $eldest\_brother_i$ - tal como no YBWC -, uma vez que a solução ótima deveria estar no intervalo da janela definida pelo processamento deste melhor nó representado por *eldest_brother*?

Se fosse considerada uma árvore perfeitamente ordenada a resposta seria sim, todavia, em árvores de jogos é difícil conseguir uma ordenação perfeita, visto que a avaliação de um nó apenas é conhecida quando ele é atingido (seção 2.3.2.3). Por esta razão, somada ao fato de que os demais nós da árvore de P_0 foram explorados em profundidades mais rasas, não há como afirmar que o *eldest_brother* é o que retem a solução ótima da busca, como ocorreria em uma árvore perfeitamente ordenada. Neste sentido, foi considerada a ideia original da versão do APHID [1] de que o retorno final do algoritmo tem boas chances de estar em torno do valor obtido na passagem i do mestre. Contudo, ao invés de utilizar apenas o valor k_i, foi utilizado os limites α_i e β_i, de modo a obter uma janela ligeiramente mais ampla.

A ideia do cálculo do valor de $diff$ - cujo intuito visou substituir a necessidade da definição empírica e manual da constante de formação da largura da janela de busca existente no APHID - foi definida a partir da observação de que a fronteira f_i sempre estará ordenada apesar das possíveis alterações que podem ocorrer a cada passagem de P_0. Logo, o nó mais a esquerda sempre estará com a melhor avaliação daquele momento i. Sendo assim, o valor obtido w_i é um forte candidato à solução final da busca, ao passo que k_i tende a ser uma solução um pouco menos precisa por não ter sido explorado no nível máximo. Nesta direção, foi criada a hipótese de que a distância entre k_i e w_i poderia criar um intervalo interessante a ser utilizado para a definição da largura da janela de busca nos escravos.

5.4 Tratamento das tarefas recebidas dos escravos

Quando um escravo P_i finaliza a exploração de seu conjunto de tarefas, ele envia um pacote ao mestre para que ele atualize as informações da borda de sua subárvore. O mestre usa esses novos valores na sua próxima varredura. Este processo é muito importante no curso da busca, uma vez que os novos valores podem alterar a ordem dos nós na borda e, consequentemente, o valor dos identificadores atribuídos a eles e que serão utilizados pela política de prioridades dos processadores escravos. Além disso, a janela de busca do mestre pode ser alterada, criando uma situação em que nós são descartados ou novos nós necessitam ser explorados. Neste caso, na AIS dos escravos haverá uma atualização na lista de tarefas que pode contemplar as seguintes situações: 1) novos nós serão acrescentados; 2) os nós que deixaram de existir na borda de busca da subárvore do mestre perderão totalmente a sua prioridade; e 3) os nós que permaneceram na borda da subárvore do mestre podem ter a sua localização alterada. Esta dinâmica de operação pode constantemente alterar a solução corrente obtida pelo mestre em sua subárvore.

Além disso, como consequência, os limites da janela de busca dos processadores escravos podem ser alterados, conforme apresentado na seção 5.3.

Considerando a importância da atividade de atualizar as informações da borda da subárvore do mestre, o ADABA implementa uma nova estratégia para tratar os resultados produzidos pelos escravos. Tal estratégia é ilustrada na Figura 37. O método *handleReceveidTasks* controla um *pool* de *threads*. Desta forma, é possível tratar um conjunto de resultados recebidos de mais de um escravo simultaneamente. O tamanho do *pool* de *threads* é definido nas configurações do sistema, uma vez que vai depender da arquitetura disponível para a execução do algoritmo.

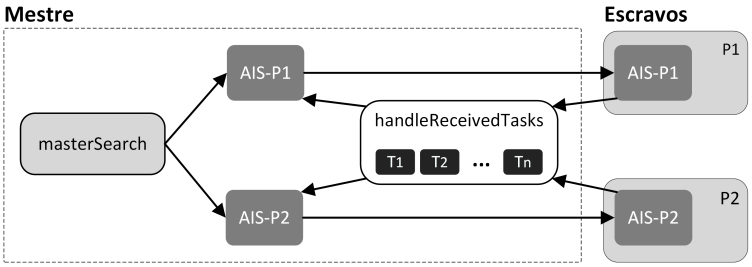


Figura 37 – Tratamento dos resultados recebidos pelos escravos no ADABA

Com esta abordagem, o processador mestre obtém mais dados recentes ao realizar uma nova varredura em sua subárvore. O efeito disso está diretamente relacionado à atualização da janela de busca e também ao controle da sobrecarga de busca, uma vez que é possível haver o estreitamento dos limites da janela de busca por meio dos resultados recebidos dos escravos.

5.5 Representação dos estados da busca no ADABA

Cada estado que é submetido à busca do ADABA é representado por uma estrutura do tipo *Node*, conforme apresentado no pseudocódigo 7:

Pseudocódigo 7 Estrutura Node

```
class Node {
int depth,
float evaluation,
boolean visited,
Board board,
Move move,
Type type,
Player player,
ScoreType scoreType,
Status status
}
```

onde no pseudocódigo 7:

- *depth:* profundidade de busca em que o nó foi explorado.
- *evaluation:* valor retornado pela função de avaliação do problema.
- *board:* representação do estado de tabuleiro do jogo, uma vez que a função de avaliação necessitará desta informação para calcular a predição associada ao nó em questão. Por exemplo, no jogo de Damas, esta estrutura poderia ser definida conforme apresentado na seção 2.5.2.1.
- *move:* armazena o movimento que deverá ser efetuado pelo jogador.
- *visited:* informa se o nó foi visitado pelo processador mestre em sua última varredura pela árvore de busca;
- *type:* indica o tipo do nó e pode assumir um dos valores listados no pseudocódigo 8, isto é, MIN (para o nó do tipo minimizador), MAX (para o nó do tipo maximizador) ou LEAF (para os nós do tipo folha).

Pseudocódigo 8 Estrutura Type

```
enum Type {
MIN, MAX, LEAF
}
```

- *player:* informa se a busca está sendo realizada na óptica do jogador Preto ou Vermelho, conforme estrutura apresentada no pseudocódigo 9.

Pseudocódigo 9 Estrutura Player

```
enum Player {
  BLACK, RED
}
```

- *scoreType:* armazena uma *flag* para indicar o real significado da predição retornada por *evaluation* podendo assumir um dos valores listados no pseudocódigo 10. Assim, *AT_MOST* indica que o valor de *evaluation* foi obtido a partir de uma poda alfa; *AT_LEAST* indica que ele foi obtido em uma poda beta; e *EXACT* indica que ele foi obtido sem a ocorrência de podas. Tal valor é importante para verificar se um nó existente na TT pode ser utilizado na busca conforme explanado na seção 5.8.1.

Pseudocódigo 10 Estrutura ScoreType

```
enum ScoreType {
  EXACT, AT_LEAT, AT_MOST
}
```

- *status:* indica a situação do nó segundo os valores apresentados no pseudocódigo 11. Nesta direção, um nó que tem o valor *UNCERTAIN* não foi explorado até a profundidade máxima da busca, ao passo que aquele que possui o valor *CERTAIN* já teve sua exploração concluída.

Pseudocódigo 11 Estrutura Status

```
enum Status {
  UNCERTAIN, CERTAIN
}
```

5.6 Comunicação entre mestre e escravos

Toda a comunicação entre mestre e escravos no ADABA é feita por meio da seguinte estrutura de dados, denominada *ADABA Intermediate Structure* (AIS):

Pseudocódigo 12 Estrutura de comunicação entre os processadores mestre e escravos no ADABA.

```
class AIS {
  int identifier,
  List<Node> tasks,
  Float alpha,
  Float beta,
  Object evaluationFunction,
  List<AIS> slaves
}
```

onde no pseudocódigo 12:

- *identifier:* indica o processador que a estrutura está representando;
- *tasks:* armazena o conjunto de tarefas atribuídas pelo mestre;
- *alpha:* indica o valor do limite inferior da janela de busca a ser utilizada pelos escravos conforme cálculo apresentado na expressão 8;

- *beta:* indica o valor do limite superior da janela de busca a ser utilizada pelos escravos conforme cálculo apresentado na expressão 9;

- *evaluationFunction:* representa um tipo genérico em que será disponibilizado o método de avaliação dos nós. Este tipo está contido na AIS pois podem haver parâmetros gerados pelo mestre que são importantes no cálculo da avaliação dos nós nos escravos. Por exemplo, no sistema do jogo de Damas apresentado no Capítulo 6, esta variável corresponde aos pesos da rede neural (seção 6.3);

- *slaves:* referencia a AIS de cada um dos escravos. É importante destacar que esta estrutura permite a criação de uma hierarquia entre os processadores. Todavia, na presente proposta do ADABA foi considerado apenas um nível, isto é, um mestre e os demais processadores escravos.

Cada escravo terá a cópia da AIS a qual é responsável. Neste contexto, quando o processador mestre P_0 desejar se comunicar com um determinado escravo P_i, ele escreverá na porção da AIS referente a este último processador e gravará uma cópia em sua AIS local. É função de P_i verificar constantemente sua AIS a fim de identificar a existência de tarefas a serem executadas. Sempre que P_i quiser retornar informações de um nó ao mestre ele o fará por meio da AIS. A Figura 38 ilustra esta comunicação entre mestre e escravos.

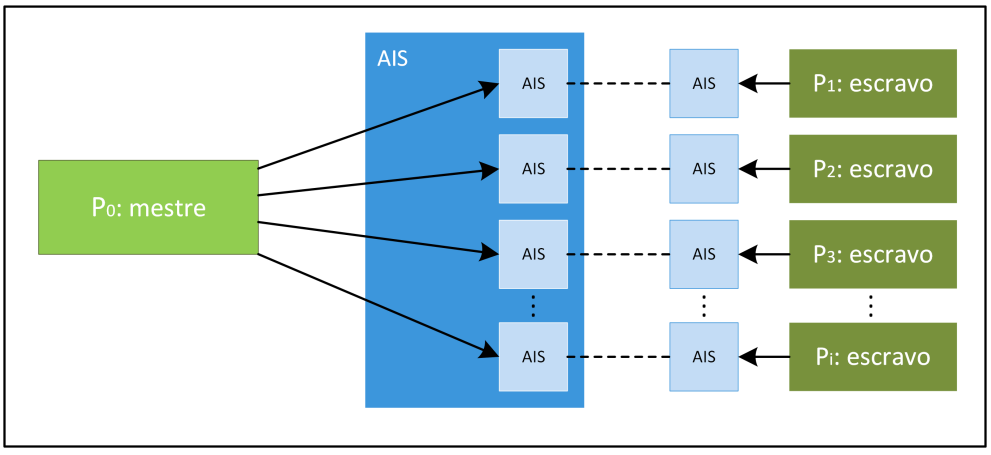


Figura 38 – Representação do canal de comunicação entre mestre e escravos no ADABA

Esta estrutura é que permite a comunicação assíncrona entre os processadores. A AIS de cada um dos processadores é criada e configurada logo no início da execução do sistema. Tal fato ocorre pois, é necessário que o mestre tenha conhecimento de todos os escravos que participarão do processo de busca. Em relação aos escravos, é importante deixa-los preparados para verifiquem se há tarefas para executar, uma vez que eles não tem conhecimento de quando serão requisitados, conforme apresentado no pseudocódigo 4.

5.7 Balanceamento de Carga

Embora o processador mestre tente distribuir a quantidade de tarefas de modo equilibrado para cada processador, nem o mestre, nem os escravos podem predizer a quantidade de esforço requerido para completar uma busca na profundidade $d - d_m$ para um dado estado. Portanto, desbalanceamento de carga pode ocorrer.

Para lidar com esta situação, o processador mestre irá checar a AIS de seus escravos a fim de verificar a quantidade de tarefas que possuem o status igual a *UNCERTAIN*. Desta forma, existe as seguintes possibilidades:

1. Não há tarefas com status igual a *UNCERTAIN*, logo significa que todas possuem status igual a *CERTAIN* e, portanto, já foram finalizadas na profundidade máxima de busca.

2. Existe apenas uma tarefa com status *UNCERTAIN*, neste caso o mestre entende que o processador escravo em questão está trabalhando nela.

3. Existe mais de uma tarefa com status igual a *UNCERTAIN*.

Desta forma, o balanceamento de carga pode ocorrer quando, após a checagem do mestre, seja verificado que há processadores que finalizaram suas tarefas (situação 1) e escravos que possuem mais de uma tarefa não finalizada (situação 3). Neste caso, o processador mestre seleciona o primeiro escravo que já finalizou suas tarefas, P_j, e o primeiro escravo que não finalizou suas tarefas, P_k, em sua lista de escravos. Desta forma, o mestre seleciona uma tarefa com status *UNCERTAIN* de P_k e a atribui a P_j.

A fim de evitar que uma tarefa seja realocada em diversos momentos, o ADABA permite que uma tarefa seja redistribuída apenas uma vez. Por exemplo, se uma tarefa t_1 sob responsabilidade de P_k for redistribuída em determinado momento a P_j, ela não poderá mais ser realocada até o final da busca. Tal situação ocorre para que os processadores possam usufruir das informações da TT e evitar reprocessamentos constantes.

5.8 Tabelas de Transposição

No ADABA é utilizado o recurso de Tabela de Transposição (TT) a fim de acelerar o processo de busca, principalmente devido à utilização da estratégia de ID (seção 2.2.2). Cada processador irá trabalhar com sua própria TT, portanto, pode-se dizer que o ADABA trabalha com TTs distribuídas.

Uma TT é implementada como uma tabela *hash*, que corresponde a uma estrutura de dados que associa *chaves* a *valores* [10]. Nela, são registrados os estados analisados no percurso da busca com seus respectivos valores de avaliação (predição). Desta forma, sempre que um estado for apresentado como entrada ao método de busca, primeiro será

verificado se ele está na TT. Caso afirmativo, tal estado poderá ser utilizado desde que satisfaça um conjunto de restrições que serão detalhadas na seção 5.8.1. Caso negativo, a busca prosseguirá sem alteração. O pseudocódigo 13 apresenta a estrutura *TranspTable* que é utilizada para armazenar cada estado S_i avaliado pelo ADABA.

Pseudocódigo 13 Estrutura da Tabela de Transposição do ADABA

```
class TranspTable{
  Key key;
  Node entry1;
  Node entry2;
}
```

Conforme é possível observar, esta estrutura é composta por um tipo *Key* (apresentado no pseudocódigo 14) e duas ocorrências do tipo *Node* (apresentado no pseudocódigo 7). Esta organização foi projetada a fim de tratar dois problemas de colisão identificados por Zobrist [72] que podem ocorrer em uma TT, denominados como erros *tipo 1* e *tipo 2*.

O erro *tipo 1*, ocorre quando dois estados distintos são mapeados na mesma chave *hash*. Caso o erro não seja detectado, pode acontecer de predições incorretas serem retornadas pelo método de busca ao consultar a TT. Para controlar esse tipo de erro, o ADABA implementa a estrutura *Key* apresentada no pseudocódigo 14, que contém duas chaves denominadas *hashvalue* e *ckecksum*. A ideia é que estas chaves sejam geradas utilizando números aleatórios independentes. Se dois estados diferentes do tabuleiro produzirem a mesma chave *hash* (*hashvalue*) é improvável que produzam, também, o mesmo valor para *checksum*. A segunda chave *hash* não precisa ser, necessariamente, do mesmo tamanho da primeira. Porém, quanto maior o número de bits presentes nas chaves *hash*, menor a probabilidade de ocorrência dos erros *tipo 1*. No ADABA, *hashvalue*, contém 64 bits e, *checksum*, possui 32 bits. No caso de jogos de tabuleiro, um meio eficiente para o cálculo e definição destas chaves é a técnica de Zobrist [72], em que são utilizadas sequências de bits aleatórios de tamanho fixo (denominados *Zobrist key*) vinculados a cada estado possível de cada um dos quadros do tabuleiro. A seção 6.2.2 apresentará como é a definição desta chave para o jogo de Damas.

Pseudocódigo 14 Estrutura da chave da TT do ADABA

```
class Key{
  long  hashvalue;
  int    checksum;
}
```

O erro *tipo 2* ocorre em virtude dos recursos finitos de memória existentes, quando dois estados distintos, apesar de serem mapeados em chaves *hash* diferentes, são direcionados para o mesmo endereço na TT. Para resolver este problema é utilizado dois esquemas de

substituição, denominados *Deep* e *New*, propostos por Breuker [73, 74]. Segundo 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.

De acordo com a estratégia de Breuker, na primeira vez que um determinado endereço end_1 é selecionado para armazenar alguma informação de um estado S_i, ele será gravado no primeiro nível *Deep*. Caso um estado S_i' seja mapeado para o mesmo endereço end_1 na TT, significa que ocorreu um erro do *tipo 2*. Neste caso, o nível *Deep* vai armazenar o estado que possuir maior profundidade. O conceito por trás desse comportamento é que uma subárvore mais profunda, normalmente, contém mais nós do que uma mais rasa, e tal fato faz com que o algoritmo de busca economize um tempo ao ser poupado de pesquisar a subárvore mais profunda. O segundo nível *New*, portanto, vai armazenar o outro estado. Se um endereço da TT armazenar em seus dois níveis informações sobre dois estados distintos do tabuleiro, então significa que o segundo nível foi utilizado para resolver um problema de colisão. Em termos práticos, o primeiro nível armazena dados de maior precisão, enquanto o segundo, age como um "cache" temporal. No ADABA, a estrutura do pseudocódigo 13 apresenta duas entradas, *entry1* e *entry2*, representando, respectivamente, os níveis *Deep* e *New*.

5.8.1 Utilização dos estados da Tabela de Transposição

Quando um estado S_i é apresentado ao ADABA, para cada um dos seus filhos c é executado o seguinte método:

$$retrieve(c, d, nodeType, besteval, bestmove); \tag{10}$$

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

Sempre que esse método é executado, verifica-se se o estado c existe na TT por meio da correspondência da chave *hash*. Considerando o caso positivo em que um estado S_i' é encontrado, os valores relacionados a *besteval* e a *bestmove* apenas poderão ser utilizados pela busca se as seguintes restrições forem satisfeitas:

1. $depth >= d$, *depth* é a profundidade de busca associada a S_i'. Essa restrição está vinculada ao fato de que quanto mais profundo os valores em que *besteval* e *bestmove* são calculados para um determinado estado, mais preciso tornam-se;

2. O valor de *scoreType* de $S_i^{'}$ é igual a *Exact*. Caso contrário, é necessário considerar que:

   - se c é do tipo minimizador, é necessário verificar se a predição de $S_i^{'}$ é menor ou igual ao limite inferior da janela de busca corrente: $S_i^{'}.prediction \leq \alpha$;
   - se c é do tipo maximizador, é necessário verificar se a predição de $S_i^{'}$ é maior ou igual ao limite superior da janela de busca corrente: $S_i^{'}.prediction \geq \beta$.

5.9 Aprofundamento Iterativo no ADABA

O ADABA trabalha naturalmente por ID (estratégia de busca apresentada na seção 2.2.2). Quando um conjunto T de tarefas são atribuídas a um escravo P_i, ele iniciará a exploração de t ($t \in T$) a partir da profundidade definida por:

$$d_{atual} = d_{last} + I; \tag{11}$$

Desta forma, a profundidade d_{atual} em que t deverá ser explorada consiste do valor correspondente ao último nível em que t foi explorada (d_{last}) adicionando o valor de incremento I. É importante salientar que, caso seja a primeira vez em que t é explorada no processador escravo, o valor de d_{last} corresponde a d_m (a profundidade da borda da subárvore do mestre). Neste sentido, para aplicar a estratégia de ID foi considerado que a busca deve ser configurada com um valor elevado para o limite de profundidade e com o tempo desejado para a realização da sugestão do movimento. Sendo assim, caso o tempo seja atingido, o *ADABA* irá interromper a busca e o mestre retornará a solução obtida até aquele momento. É importante ressaltar que neste caso pode ocorrer de diferentes nós da fronteira da subárvore do mestre conterem valores de avaliações obtidos em diferentes profundidades da busca. Todavia, isto não é uma limitação, visto que o algoritmo terá explorado o tempo máximo proporcionado de modo a atingir o melhor $look - ahead$ possível. Caso o valor configurado para a profundidade máxima da busca seja atingido, a busca será finalizada (segundo a estratégia de profundidade limitada).

Ressalta-se que na técnica ID convencional - apresentada na seção 2.2.2 -, são realizadas uma série de buscas partindo da raiz da árvore e expandindo até um determinado nível. A cada iteração haverá um incremento neste nível até que o limite de tempo seja atingido. Esta abordagem não é interessante para o ADABA, pois como a busca é reiniciada, existirá novos envios das mesmas tarefas aos escravos, o que acarretará em maior sobrecarga de comunicação.

5.10 Considerações Finais

Este Capítulo apresentou a proposta do ADABA, uma nova versão distribuída do Alfa-Beta que segue a abordagem assíncrona de paralelismo. Tal versão foi inspirada no modelo mestre-escravo do APHID. Neste contexto, o ADABA visou atacar algumas fragilidades identificadas no APHID por meio das seguintes propostas: inclusão de uma política automatizada para formar e atualizar a janela de busca nos processadores escravos inspirada nas ideias do algoritmo síncrono YBWC; tratamento das soluções enviadas pelos escravos por meio de um grupo de *threads* inerentes ao processo mestre; e implementação de uma nova política de prioridade de seleção das tarefas que os escravos devem explorar.

O ADABA será a base do mecanismo de tomada de decisão dos agentes jogadores de Damas construídos no âmbito deste trabalho, a saber: o monoagente ADABA-Draughts (Capítulo 6) e o multiagente D-MA-Draughts (Capítulo 7). Além disso, o desempenho do algoritmo será avaliado em experimentos apresentados no Capítulo 8.

Capítulo 6

ADABA-Draughts

Este capítulo se refere ao objetivo específico 4 relacionado na seção 1.3.1. Desta forma, será apresentado o ADABA-Draughts: um sistema monoagente jogador de Damas cujo mecanismo de tomadas de decisão é composto pelo algoritmo de busca ADABA (detalhado no Capítulo 5). A finalidade do ADABA-Draughts é permitir a validação do desempenho do algoritmo proposto neste trabalho de doutorado em um problema de alta complexidade.

As seções deste capítulo estão organizadas como se segue: 6.1 apresenta a arquitetura geral do sistema jogador ADABA-Draughts; a seção 6.2 apresenta os detalhes do módulo de busca do sistema; a seção 6.3 apresenta o processo de treinamento ADABA-Draughts; e, por fim, a seção 6.4 apresenta as considerações finais.

6.1 Arquitetura Geral do ADABA-Draughts

A Figura 39 ilustra a arquitetura geral e o ciclo de operações executadas pelo agente ADABA-Draughts a cada vez que uma tomada de decisão para a escolha de um movimento a partir do estado corrente do jogo é efetuada. Ressalta-se ele possui os seguintes tipos de jogo:

1. *Treino*: trata-se dos jogos compreendidos na etapa de treinamento da MLP;
2. *Disputa*: trata-se dos jogos em que o ADABA-Draughts está apto a jogar uma partida contra um oponente.

É importante ressaltar que, em função do tipo de jogo, o ciclo da Figura 39 apresenta comportamento particular para os jogos de disputa em que somente os passos de #1 a #6 são executados. Neste aspecto, este ciclo compreende:

Passo #1 : O estado corrente do tabuleiro (chamado aqui T_x) é apresentado ao *Módulo de Busca*. Assim sendo, T_x foi obtido a partir da execução do último movimento

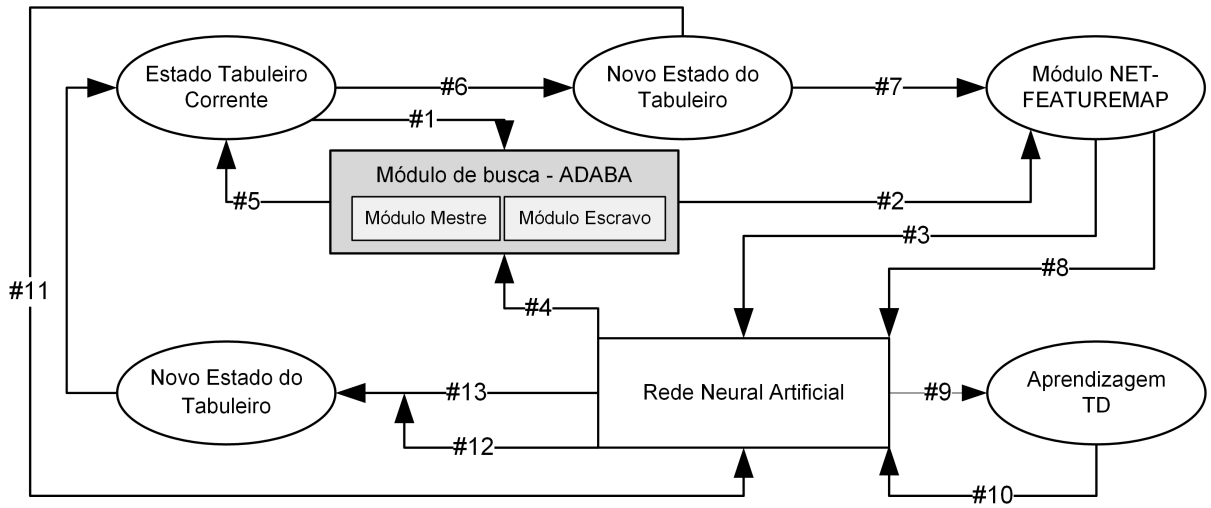


Figura 39 – Arquitetura geral do jogador ADABA-Draughts

m_x. Além disso, P_x representa a avaliação produzida pela MLP para T_x no ciclo anterior do treinamento.

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

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

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

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

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

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

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

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

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

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

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

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

Módulo de Busca: responsável pela tomada de decisão do agente em função do estado corrente do tabuleiro por meio do algoritmo de busca ADABA. Tal estado é recebido na representação vetorial.

Rede Neural Multicamadas: a rede neural recebe, em sua camada de entrada, um estado de tabuleiro do jogo e devolve, em seu único neurônio na camada de saída, um valor real compreendido entre -1.0 e +1.0. Tal valor (predição) representa o quão o estado do tabuleiro presente na camada de entrada da rede é favorável ao jogador automático;

Módulo NET-FEATUREMAP: por meio do mapeamento das características utilizadas pelo sistema, esse módulo converte a representação vetorial do estado corrente que lhe é repassado pelo módulo de busca em representação NET-FEATUREMAP, apresentando esta última à entrada da rede neural MLP para avaliação. O ADABA-Draughts utiliza 14 características para representação dos estados do tabuleiro em seu mapeamento NET-FEATUREMAP (conforme descrito na seção 6.3). Desta forma, o mapeamento C é C:T $\rightarrow \mathbb{N}^{14}$.

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

As seções a seguir apresentam os detalhes do Módulo de Busca (seção 6.2) e do Módulo de Aprendizagem (seção 6.3), o qual envolve a descrição dos Módulo NET-FEATUREMAP e Rede Neural Multicamadas.

6.2 Módulo de Busca

O módulo de busca do ADABA-Draughts é composto pelo algoritmo distribuído ADABA - apresentado no Capítulo 5 - aliado a TT empregando uma das estratégias de busca: Profundidade Limitada ou ID. Neste sentido, a seção 6.2.1 apresenta a infraestrutura utilizada para executar o algoritmo de busca ADABA, e na sequência, a seção 6.2.2 apresenta detalhes da TT utilizada no ADABA-Draughts.

6.2.1 Infraestrutura para execução do algoritmo ADABA

Para a execução do algoritmo de busca, o ADABA trabalha com um número configurável m de processadores. Cada um deles é identificado por um número inteiro i denotado como P_i $(0 \leq i < m)$. O processador P_0, denominado mestre, irá atuar em todos os módulos do jogador (conforme arquitetura ilustrada na Figura 39). Os demais processadores P_i $(0 < i < m)$, denominados escravos, irão atuar apenas no Módulo de Busca. A Figura 40 ilustra a infraestrutura em que o ADABA-Draughts foi implementado.

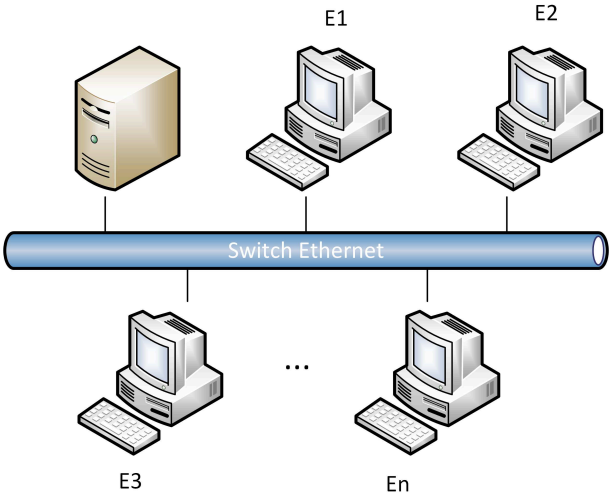


Figura 40 – Infraestrutura do jogador ADABA-Draughts

Como é possível observar, trata-se de um ambiente de memória distribuída composto por um servidor que contem os módulos mestre e escravo. Os demais elementos da infraestrutura correspondem a n estações que processam apenas o módulo escravo. A composição deste ambiente é configurada em um arquivo de propriedades[1].

[1] O ADABA-Draughts possui um arquivo de propriedades que relaciona todas as configurações do sistema.

Um processador pode assumir o papel de mestre ou escravo, mas não ambos. A Figura 41(a) apresenta os possíveis estados de um processador mestre e a Figura 41(b) os estados de um processador escravo.

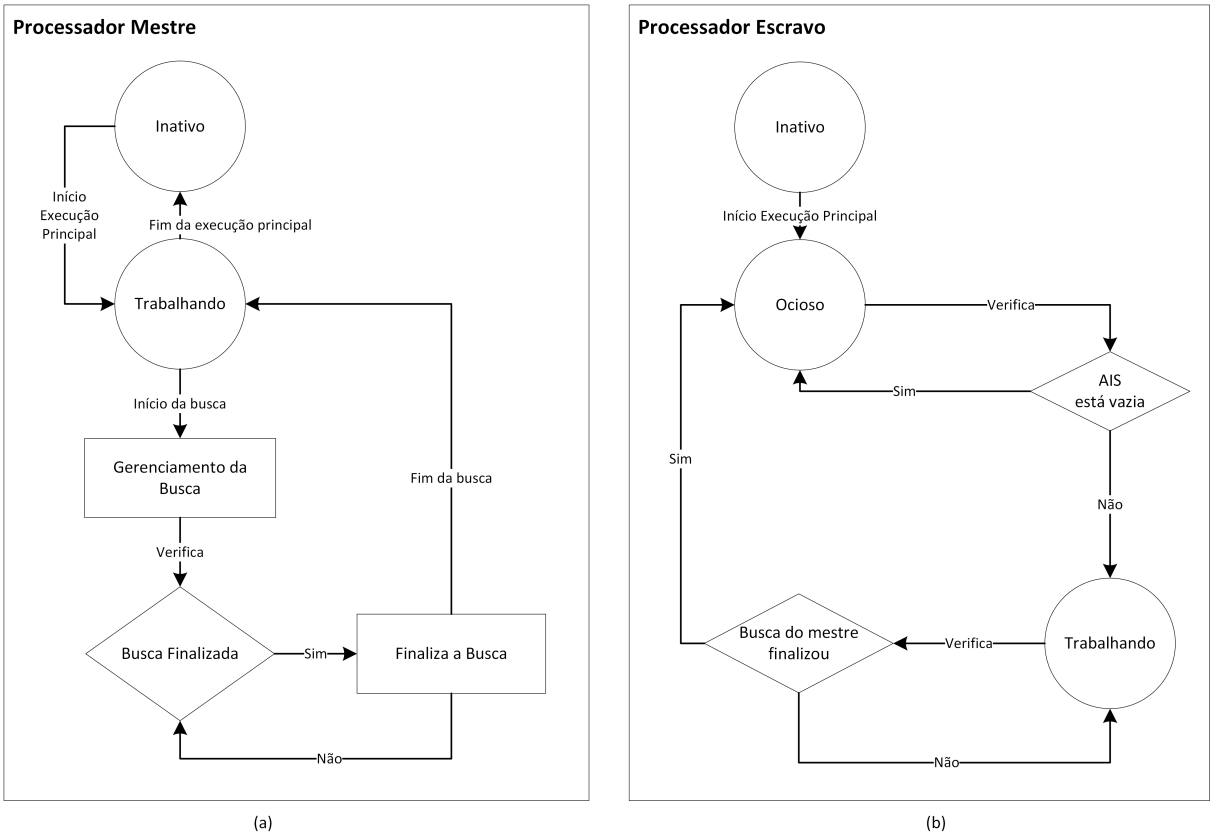


Figura 41 – Estados dos processadores no ADABA-Draughts: (a) processador mestre; (b) processador escravo

O processador mestre estará no estado *Inativo* até que o sistema ADABA-Draughts inicie sua participação em uma partida. Portanto, quando isto ocorre, o mestre passará ao estado *Trabalhando*, pois ele atua em todos os módulos da arquitetura do jogador ADABA-Draughts apresentada na Figura 39. Sempre que for necessária uma tomada de decisão para indicar uma ação (movimento) para o ADABA-Draughts, o mestre iniciará o processo de *Gerenciamento da Busca* (detalhado na seção 5.1.1). O mestre permanecerá neste estado até que a busca finalize. Neste caso, ele notificará os processadores escravos sobre o encerramento da busca e, na sequência, voltará a atuar nos demais módulos do ADABA-Draughts. Desta forma, tal processador apenas finalizará suas atividades quando o sistema ADABA-Draughts for encerrado, neste caso, ele retorná ao estado *Inativo*.

Um processador escravo também é iniciado junto à execução do sistema ADABA-Draughts, todavia, ele assume o estado *Ocioso*, uma vez que não há tarefas atribuídas a ele. Apesar de estar no estado *Ocioso*, o escravo verifica constantemente a existência de tarefas em sua estrutura AIS (apresentada na seção 5.6). Neste contexto, se a AIS estiver

vazia, o escravo permanece no estado *Ocioso*. Por outro lado, se for constatado a existência de tarefas, ele passará ao estado *Trabalhando* e irá executar todas as atividades descritas na seção 5.1.2 até que seja notificado pelo mestre para encerrar suas atividades, pois a busca principal finalizou. Neste caso, o processador limpa todos os recursos utilizados até o momento e retorna ao estado *Ocioso* para aguardar novas informações que ocorrerão quando o mestre iniciar um novo processo de tomada de decisão.

6.2.2 Tabela de Transposição: Criação de Chaves Hash com a Técnica de Zobrist

Conforme apresentado na seção 5.8, no ADABA uma TT é implementada como uma tabela *hash*. Por esta razão, é necessário a utilização de um método que permita a criação de chaves *hash* que identifiquem um estado de tabuleiro. Nesta direção, o método de Zobrist [72] é muito eficiente para o caso de jogos de tabuleiro. Este método utiliza o operador **XOR** (ou exclusivo), simbolizado matematicamente por $\otimes$. Logicamente, o **XOR** é um tipo de disjunção lógica entre dois operandos que resulta em "verdadeiro" se, e somente se, exatamente, um dos operandos tiver o valor "verdadeiro", caso contrário, a operação resulta em "falso". Computacionalmente, o operador **XOR** pode ser aplicado sobre dois operandos numéricos [51]. Neste caso, os operadores "verdadeiro" e "falso" correspondem, respectivamente, aos bits "1" e "0". Por exemplo:

1. operandos numéricos na base binária: o **XOR** aplicado sobre dois bits quaisquer resulta em "1" se, e somente se, exatamente, um dos operandos tiver o valor "1". Assim, considerando as sequências binárias de n bits $Seq_1 = b_1, b_2, ..., b_n$ e $Seq_2 = r_1, r_2, ..., r_n$, para calcular $Seq_3 = Seq_1 \otimes Seq_2$, basta aplicar o operador **XOR** sobre os bits das posições correspondentes de Seq_1 e Seq_2, isto é, basta fazer $Seq_3 = b_1 \otimes r_1, b_2 \otimes r_2, ..., b_n \otimes rn$;

2. operandos numéricos na base decimal: a operação **XOR** sobre dois inteiros decimais segue o mesmo procedimento mostrado para operandos numéricos na base binária, sendo que, neste caso, os dois argumentos inteiros decimais devem ser, antes de tudo, convertidos para a base binária.

O operador **XOR** aplicado sobre sequências aleatórias (r) de inteiros decimais de n bits respeita as seguintes propriedades [72]:

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

5. s_i é uniformemente distribuída (uma variável é dita uniformemente distribuída quando assume qualquer um dos seus valores possíveis com a mesma probabilidade).

Supondo que exista um conjunto finito S qualquer e que se deseje criar chaves *hash* para os subconjuntos de S. Um método simples seria associar inteiros aleatórios de n bits aos elementos de S e, a partir daí, definir a chave *hash* de um subconjunto S_0 de S como sendo o resultado da operação $\otimes$ sobre os inteiros associados aos elementos de S_0. Pelas propriedades 1 e 2, a chave *hash* é única e, pelas propriedades 4 e 5, a chave *hash* é aleatória e uniformemente distribuída. Se qualquer elemento for adicionado ou retirado do subconjunto S_0, a chave *hash* mudará pelo inteiro que corresponde àquele elemento. No caso do ADABA-Draughts, existem 2 tipos distintos de peças (peça simples e rainha), 2 cores distintas de peças (peça preta e peça vermelha) e 32 casas no tabuleiro do jogo. Então, existem, no máximo, 128 possibilidades distintas (2x2x32) de colocar alguma peça em alguma casa do tabuleiro. Assim, criou-se um vetor de 128 elementos inteiros aleatórios, mostrado na Figura 42, para representar os estados possíveis do tabuleiro (cada elemento representa uma possibilidade de se ocupar uma das 32 casas do tabuleiro com alguma das 4 peças inerentes ao jogo). A chave *hash*, para representar cada estado, é o resultado da operação **XOR** realizada entre todos os elementos do vetor associados às casas não vazias do tabuleiro. Na Figura 42, pode-se visualizar que são criadas 4 chaves para cada estado possível, o que cobre a necessidade de variações de chaves em um jogo de Damas.

A técnica de Zobrist é, provavelmente, o método disponível mais rápido para calcular uma chave *hash* associada a um estado do tabuleiro do jogo de damas [72]. Isto devido à velocidade com que se executa a operação **XOR** por uma CPU. Para entender como ocorre a atualização incremental dessas chaves *hash*, considera-se as movimentações possíveis no jogo de damas:

1. *movimento simples*: pode ser tratado como a remoção de uma peça simples da casa de origem do movimento e sua inserção na casa de destino do movimento;

2. *promoção*: ocorre quando uma peça simples se torna rainha. Pode ser tratada como sendo a remoção de um tipo de peça e a inserção de outro tipo de peça na mesma casa do tabuleiro;

3. *captura*: pode ser tratada como sendo a remoção da peça capturada, a remoção da peça capturadora da casa de origem do movimento e a inserção da peça capturadora na casa de destino do movimento.

Considerando o exemplo de movimento simples da Figura 43 e o vetor de números aleatórios utilizados pelo ADABA-Draughts, mostrado na Figura 42:

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

Figura 42 – Vetor de 128 elementos inteiros aleatórios utilizados pelo ADABA-Draughts

1. para conseguir o número aleatório associado à peça preta simples, localizada na casa 21 do tabuleiro S_0, basta fazer uma consulta ao vetor V e encontrar o valor $I_{21} = 1171196056361380757$;

2. para conseguir o número aleatório associado à peça preta simples, localizada na casa 22 do tabuleiro S_0, basta fazer uma consulta ao vetor V e encontrar o valor $I_{22} = 9204715365712158256$;

3. para conseguir o número aleatório associado à peça branca simples (que também poderia ser chamada de peça vermelha), localizada na casa 31 do tabuleiro S_0, basta

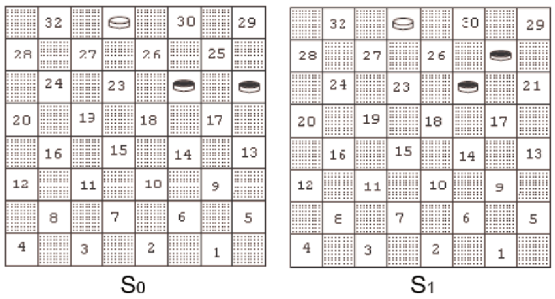


Figura 43 – Exemplo de movimento simples.

fazer uma consulta ao vetor V e encontrar o valor $I_{31} = 6271745259985944523$;

4. para conseguir a chave hash C_0, associada ao estado do tabuleiro S_0, basta considerar $C_0 = I_{21} \otimes I_{22} \otimes I_{31}$. Logo, $C_0 = 1171196056361380757 \otimes 9204715365712158256 \otimes 6271745259985944523$, ou seja, $C_0 = 4104165011015584366$;

5. para conseguir a chave hash C_1, associada ao estado do tabuleiro S_1, não é necessário repetir os passos anteriores; basta atualizar, incrementalmente, o valor da chave C_0 conforme passos 6 e 7;

6. $C_1 = C_0 \otimes I_{21}$. Indica a remoção da peça preta simples da casa 21 do estado S_0 (conforme a propriedade número 3). Logo, $C_1 = 4104165011015584366 \otimes 1171196056361380757$, ou seja, $C_1 = 2932970144385327611$;

7. $C_1 = C_1 \otimes I_{25}$. Isso indica a inserção da peça preta simples na casa 25 do tabuleiro, gerando o estado S_1 (conforme a propriedade número 1). Logo, $C_1 = 2932970144385327611 \otimes 8978665553187022367$, ou seja, $C_1 = 8988798927364941823$.

6.3 Módulo de Aprendizagem

O Módulo de Aprendizagem do ADABA-Draughts consiste no ajuste de pesos da rede MLP jogadora que avalia os estados de tabuleiro gerados pelo algoritmo de busca ADABA. A atualização dos pesos é feita através do método TD(λ) (seção 2.7). Conforme visto no ciclo de treinamento do ADABA-Draughts na Figura 39, os tabuleiros avaliados pela MLP é representado por características (*features*) do próprio jogo de Damas. O módulo responsável por esta conversão é o NET-FEATUREMAP, que recebe como entrada a representação vetorial (seção 2.5.2.1) do estado corrente do tabuleiro do jogo e o converte para a representação por *features* (seção 2.5.2.2).

O resultado do mapeamento NET-FEATUREMAP constitui os neurônios da camada de entrada da MLP, conforme apresentado na Figura 44 que ilustra um estado de tabuleiro arbitrário representado por NET-FEATUREMAP na entrada da MLP que é composta por N_A neurônios. Tais neurônios são usados para representar os bits correspondentes

ao conjunto de *features*. Cada característica tem um valor absoluto que representa sua medida analítica em um determinado estado do tabuleiro. Esse valor é convertido em bits significativos, ou seja, conversão numérica entre as bases decimal e binária, onde em conjunto com os demais bits das outras características presentes na conversão do mapeamento NET-FEATUREMAP constituem a sequência binária a ser representada na entrada da rede neural. Desta forma, o número de neurônios que irão compor a entrada da rede MLP é definido pela soma de todos os bits representativos das características que formam o mapeamento NET-FEATUREMAP.

O ADABA-Draughts utiliza as características apresentadas na Tabela 4 (os detalhes de cada uma destas *features* podem ser visualizados na seção 2.5.2.2), logo N_A é igual a 41. Por exemplo, considerando a Figura 44, existe apenas duas *features* ativas (F1 e F14) na entrada atual relacionada ao estado corrente do tabuleiro. Assim, os primeiros quatro neurônios de entrada serão usados para representar as peças neste tabuleiro que satisfaz F1 e os três últimos neurônios de entrada irão representar o número de peças, no mesmo tabuleiro, que satisfaz F14. A camada oculta da MLP possui 20 neurônios e a camada de saída é composta por um neurônio.

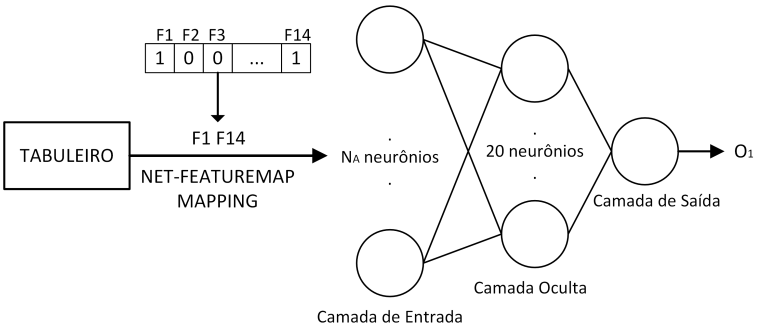


Figura 44 – Ilustração do mapeamento NET-FEATUREMAP na entrada da RNA

As próximas seções apresentam maiores detalhes do processo de treinamento da MLP do ADABA-Draughts. Neste sentido, a seção 6.3.1 explica como é realizado o cálculo da predição (avaliação) do estado de tabuleiro apresentado à MLP para a escolha da melhor ação; a seção 6.3.2 apresenta como é realizado o reajuste de pesos na MLP; e a seção 6.3.3 descreve o processo de treinamento por *self-play* com clonagem no ADABA-Draughts.

6.3.1 Cálculo da Predição e Escolha da Melhor Ação

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

1. a representação interna do tabuleiro S_t é mapeada na entrada da rede neural pelo módulo de mapeamento NET-FEATUREMAP mostrado na Figura 39;

Características	Nº Bits
F1: *PieceAdvantage*	4
F2: *PieceDisadvantage*	4
F3: *PieceThreat*	3
F4: *PieceTake*	3
F5: *Advancement*	3
F6: *DoubleDiagonal*	4
F7: *BackRowBridge*	1
F8: *CentreControl*	3
F9: *XCentreControl*	3
F10: *TotalMobility - MOB*	4
F11: *Exposure*	3
F12: *KingCentreControl*	3
F13: *DiagonalMoment*	3
F14: *Threat*	3

Tabela 4 – Conjunto de Características implementadas no jogador ADABA-Draughts

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

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

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

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

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

Funcionalmente, predições P_t's calculadas pela MLP podem ser vistas como uma estimativa do quão o estado S_t se aproxima de uma vitória do agente (representada pelo

retorno do valor +1 pelo ambiente), derrota do agente (representada pelo retorno do valor -1 pelo ambiente) ou empate (representado pelo retorno do valor 0, ou próximo de 0, pelo ambiente). Assim, configurações de tabuleiros (ou estados do jogo) que receberem predições próximas de +1 tenderão a ser consideradas como bons estados de tabuleiro, resultantes de boas ações, que poderão convergir para vitória (+1). Da mesma forma, tabuleiros cujas predições estão próximas de -1 tenderão a ser considerados péssimos estados de tabuleiro, resultantes de ações ruins, que poderão convergir para derrota (-1). O mesmo vale para configurações de tabuleiros próximos de 0, que poderão convergir para empate (0 ou valor próximo deste).

6.3.2 Reajuste de Pesos da Rede Neural MLP

O reajuste dos pesos da rede é *online*, isto é, o agente vai jogando contra o seu oponente e os pesos da rede vão sendo ajustados pelo método TD(λ) a cada vez que o agente executa uma ação (um movimento). Considerando um conjunto de movimentos (análogo a um conjunto de ações efetuadas por um agente sobre o ambiente, conforme apresentado na seção 2.7) que o agente executa durante todo o jogo $\{M_0, ..., M_{i-1}, M_i, M_{i+1}, ..., M_t\}$, onde M_t se refere a um estado final de um episódio (aqui um estado final do jogo), haverá ajuste de pesos nas seguintes circunstâncias:

- após a execução de cada movimento intermediário M_t (em que M_t é distinto do movimento de final do episódio M_f) sugerido a partir de um estado S_t. Neste caso, os pesos serão calculados com base nos valores de predição de sucesso produzidos pela rede neural.

- após a execução do final do episódio, isto é, do movimento M_f. Neste caso, os ajustes dos pesos serão calculados com base no desempenho final obtido pelo agente, ou seja, com base nos valores (+1) ou (-1) retornados pelo ambiente nos casos de vitória ou de derrota, respectivamente.

Basicamente, o agente jogador seleciona a melhor ação M_t a ser executada a partir de um estado S_t com o auxílio do algoritmo de busca ADABA e dos pesos atuais da rede neural. O estado S_{t+1} resulta da ação M_t sobre o estado S_t. A partir de então, o estado S_{t+1} é mapeado na entrada da rede neural e tem sua predição P_{t+1} calculada (a predição é o valor de saída no neurônio da última camada da rede neural). Os pesos da rede neural são reajustados com base na diferença entre P_{t+1} e a predição P_t (calculada no ciclo anterior para o estado S_t). Após o fim de cada partida de treino, um reforço final é fornecido pelo ambiente informando o resultado obtido pelo agente jogador em função da sequência de ações que executou (+1 para vitória, -1 para derrota e um valor próximo de 0 para empate).

Formalmente, o cálculo do reajuste dos pesos é definido pela equação do método TD(λ) de Sutton [59]:

$$\begin{aligned} w_{ij}^{(l)} &= w_{ij}^{(l)}(t) + \Delta w_{ij}^{(l)}(t+1) \\ &= w_{ij}^{(l)}(t) + \alpha^{(l)}(P_{t+1} - P_t) \sum_{k=1}^{t+1} (\lambda^{(t+1)-k} \nabla w P_k) \qquad (12) \\ &= w_{ij}^{(l)}(t) + \alpha^{(l)}(P_{t+1} - P_t) elig_{ij}^{(l)}(t) \end{aligned}$$

onde:

- $\alpha^{(l)}$ 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 na camada l e a entrada do neurônio j na camada $l+1$ no instante temporal t. A correção aplicada a esse peso no instante temporal $t+1$ é representado por $\Delta w_{ij}^{(l)}(t+1)$;
- O termo $elig_{ij}^{(l)}(t)$ é único para cada peso sináptico $w_{ij}^{(l)}$ 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 instante temporal t. O rastro de eligibilidade é um dos mecanismos básicos na AR para lidar com recompensas "atrasadas" (do inglês *delayed rewards*). Mais detalhes sobre o rastro de elegibilidade pode ser encontrado em [75];
- $\nabla w P_k$ representa a derivada parcial de P_k em relação aos pesos da rede no instante k. Cada predição P_k é uma função dependente do vetor de entrada $\overrightarrow{X(k)}$ e do vetor de pesos $\overrightarrow{W(k)}$ da rede neural no instante temporal k;
- O termo $\lambda^{(t-k)}$, para $(0 \leq \lambda \leq 1)$, tem o papel de dar uma "pesagem exponencial" para a taxa de variação das predições calculadas a k passos anteriores de t. 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 reajustes dos pesos por Diferenças Temporais TD(λ) são descritos nas seguintes etapas:

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

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

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

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

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

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

     onde λ tem o papel de dar uma "pesagem exponencial" para a taxa de variação das predições calculadas a k passos anteriores de t;$a_i^{(l)}$ o sinal de saída do neurônio i na camada l;$g'(x) = (1 - x^2)$ representa a derivada da função de ativação (tangente hiperbólica) [65].

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

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

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

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

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

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

$$\Delta w_{ij}^{(l)}(t+1) = \alpha^{(l)}.e(t+1).elig_{ij}^{(l)}(t), \tag{13}$$

onde o parâmetro de aprendizagem $\alpha^{(l)}$ é definido por Caixeta [51] 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 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 μ para tentar solucionar este 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 subótimo [65]. Para resolver esse problema foi empregado uma checagem de direção na equação 12, ou seja, o termo 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 da camada l, para $0 \leq l \leq 1$, é definida por:

$$w_{ij}^{(l)}(t+1) = w_{ij}^{(l)}(t) + \Delta w_{ij}^{(l)}(t+1); \tag{14}$$

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

- ❏ calcule $\Delta w_{ij}^{(l)}(t+1)$ pela equação 12;
- ❏ 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:

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

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

A estratégia de treino adotada pelo ADABA-Draughts é por *self-play* com clonagem (seção 2.8) e segue os seguintes passos:

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

2. antes de iniciar qualquer treinamento, a rede *opp1* é clonada para gerar a rede clone *opp1-clone*;

3. inicia-se então o treinamento da rede *opp1* que joga contra a rede *opp1-clone* (oponente). As redes começam uma sequência de n jogos de treinamento. Somente os pesos da rede *opp1* são ajustados durante os n jogos;

4. ao final dos n jogos de treinamento, um torneio de dois jogos é realizado para verificar qual das redes é a melhor. Caso o nível de desempenho da rede *opp1*

supere o nível da rede *opp1-clone*, é realizada a cópia dos pesos da rede *opp1* para a rede *opp1-clone*. Caso contrário, não se copiam os pesos e a rede original *opp1* permanece inalterada para a próxima sessão de treinamento;

5. o passo 3 é retomado de modo que é executada uma nova sessão de n jogos de treinamento entre a rede *opp1* e o seu último clone *opp1-clone*. Este processo se repete até que um número máximo de jogos de treinamento (parâmetro do jogo) seja alcançado.

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

6.4 Considerações Finais

Este capítulo apresentou o sistema monoagente jogador de Damas ADABA-Draughts que atua em ambiente de alto desempenho a partir do algoritmo ADABA desenvolvido no âmbito deste trabalho. A arquitetura do ADABA-Draughts será a base para os testes de desempenho do algoritmo proposto neste trabalho, conforme será apresentado nos experimentos do Capítulo 8. Além disso, este monoagente constituíra a arquitetura individual dos agentes do SMA jogador de Damas proposto neste trabalho, o D-MA-Draughts, a ser apresentado no próximo Capítulo.

Capítulo 7

D-MA-Draughts - Inserindo o ADABA em uma plataforma multiagente

Este capítulo se refere ao cumprimento de dois objetivos específicos apresentados na seção 1.3.1, a saber: objetivo 1, que trata da identificação da dinâmica de jogo mais adequada a produzir bons resultados em partidas no SMA D-MA-Draughts [3]; e objetivo 5 que se refere à inserção da arquitetura do monoagente ADABA-Draughts como arquitetura individual dos agentes do D-MA-Draughts. Ressalta-se que a autora do presente trabalho já implementou uma primeira versão do D-MA-Draughts em [3] que emprega o algoritmo distribuído síncrono YBWC como mecanismo de tomada de decisão. Uma breve descrição deste agente jogador foi apresentada na seção 3.2.2.5 de trabalhos correlatos.

Neste contexto, salienta-se que a primeira versão do D-MA-Draughts possui duas dinâmicas de atuação em partidas que se diferem no modo de cooperação entre os agentes em fases finais do jogo. Todavia, em Ref. [3] não foi realizada uma análise sobre qual dinâmica pudesse melhor representar o D-MA-Draughts. Sendo assim é fundamental definir a melhor dinâmica de atuação do jogador para que neste trabalho seja adotada aquela que produzir melhores resultados em partidas.

Além disso, considerando o objetivo geral deste trabalho que contempla a criação de um novo algoritmo distribuído de busca baseado no Alfa-Beta, que culminou na criação da arquitetura monoagente ADABA-Draughts, pretende-se melhorar o desempenho geral do D-MA-Draughts ao inserir este novo algoritmo, por meio do ADABA-Draughts, na arquitetura individual dos agentes.

Este capítulo está organizado da seguinte forma: a seção 7.1 apresenta a arquitetura geral do D-MA-Draughts; a seção 7.2 mostra como foi o procedimento de obtenção dos agentes de final de jogo; a seção 7.3 apresenta as dinâmicas de final de jogo do D-MA-Draughts que são o foco das análises deste capítulo; a seção 7.4 apresenta a arquitetura individual dos agentes; e a seção 7.5 apresenta as considerações finais sobre esta etapa da pesquisa.

7.1 Arquitetura do D-MA-Draughts

A arquitetura geral do D-MA-Draughts é ilustrada na Figura 45 e apresenta os seguintes módulos:

IIGA: trata-se de um agente que é treinado para ser especialista em fases iniciais e intermediárias de jogo, o que lhe acarreta a responsabilidade de conduzir a partida em tabuleiros com, no mínimo, 13 peças.

Redes Kohonen-SOM: este módulo possui duas finalidades:

1. *Clusterização de estados de tabuleiro de final de jogo segundo suas similaridades*: aqui a rede Kohonen-SOM é treinada para ser responsável por agrupar milhares de tabuleiros de final de jogo presentes em uma base de dados obtidos em jogos reais. Após treinada, a rede Kohonen-SOM deve ser capaz de apontar a qual dos grupos gerados pertence um determinado estado de tabuleiro de final de jogo, baseando-se, para tanto, na similaridade deste tabuleiro com cada grupo (ele pertencerá àquele do qual for mais próximo). Conforme será visto na seção 7.2, para a base de dados adotada o número de *clusters* obtido foi de 25. Para cada *cluster* do D-MA-Draughts é associada uma rede MLP que será treinada de forma a se especializar no *perfil* de tabuleiro de final de jogo do *cluster* que ela representa. Tais MLPs correspondem aos agentes de final de jogo. Assim sendo, haverá 25 desses agentes.
2. *Classificação do End Game Agent (EGA) dentre os agentes de final de jogo*: neste caso, a rede deverá definir, dentre os 25 agentes de final de jogo aquele mais apto a conduzir a partida quando esta atingir um estado de final de jogo.

Agentes de Final de Jogo: conforme dito acima, esse módulo consiste de 25 redes MLPs jogadoras. Cada uma delas será treinada baseando-se nos estados de tabuleiros presentes em cada *cluster* correspondente.

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

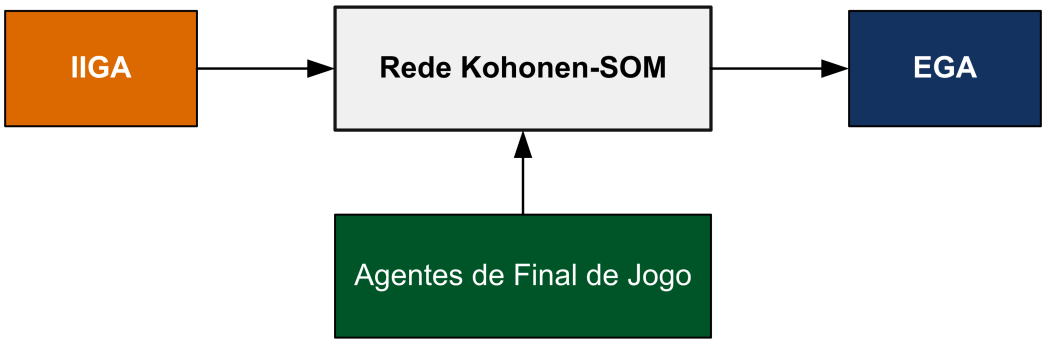


Figura 45 – Arquitetura do jogador D-MA-Draughts

7.2 Processo de Geração dos Agentes de Final de Jogo

Em Damas, o jogo começa a assumir aspectos de desfecho de partida quando o tabuleiro tem, no máximo, 12 peças. Portanto, quando o tabuleiro atinge esta quantidade de peças, o jogo passa para uma fase denominada de final de jogo.

No jogo de Damas existem cerca de um quadrilhão de tabuleiros que define uma fase de final de jogo [62]. Este fato torna impraticável o tempo de treinamento requerido à grande quantidade de MLPs que seriam necessárias para representar este grande número de tabuleiros. No entanto, apesar de existir uma quantidade muito elevada de tabuleiros de final de jogo, alguns deles raramente ocorrem em partidas reais, como por exemplo, tabuleiros com 12 Damas. Considerando este fato e com o objetivo de obter os estados de tabuleiro mais recorrentes, as redes MLP do D-MA-Draughts foram geradas a partir de uma base composta de estados de tabuleiro obtidos de partidas reais envolvendo os seguintes jogadores automáticos: NeuroDraughts [65], LS-Draughts [76] e VisionDraughts [51]. No geral, 5000 jogos foram executados. Destes, 4000 dos estados de tabuleiro mais recorrentes foram selecionados. Tais estados compuseram a base de dados utilizada pelo D-MA-Draughts no processo de clusterização. Neste processo, os estados foram agrupados de modo a constituir os perfis aos quais as redes MLPs correspondentes aos agentes de final de jogo seriam treinadas. Para lidar com esta tarefa, foi utilizada uma rede Kohonen-SOM. A Figura 46 apresenta a arquitetura geral do processo de clusterização da base de dados do D-MA-Draughts. Os módulos presentes neste processo são:

- **Base de Dados**: composta por 4000 estados de tabuleiro de final de jogo. Cada um destes estados é representado como um vetor (seção 2.5.2.1).

- **NET-FEATUREMAP**: este módulo é responsável por converter a representação vetorial de um estado de tabuleiro para a representação por *features* (representação NET-FEATUREMAP, seção 2.5.2.2).

- **Kohonen-SOM**: este módulo tem como entrada a nova representação do estado de tabuleiro produzida no módulo NET-FEATUREMAP (correspondendo às entradas dos neurônios X_1, ..., X_j na Figura) e como saída os neurônios Y_1, ..., Y_{25}, que representam os *clusteres* a que cada estado deve se direcionar. Mais especificamente, durante o processo de clusterização cada estado de final de jogo da base de dados é submetido a entrada da rede Kohonen-SOM para que a rede aponte o *cluster* que este estado de tabuleiro deve pertencer (que é definido pelo neurônio de saída correspondente). Este processo é baseado no cálculo da Distância Euclidiana entre os neurônios de entrada e cada neurônio da camada de saída.

- **Clusterização**: estes *clusters* correspondem a 25 bases de dados que representam as saídas da rede Kohonen-SOM. Cada neurônio da camada de saída é conectado a um *cluster* particular. No momento em que a Kohonen-SOM aponta o neurônio de saída, o vetor baseado em representação vetorial é armazenado no *cluster* vinculado a ele. Os estados de tabuleiro que pertencem a um determinado *cluster*, apesar de serem distintos entre si, satisfazem um critério mínimo de similaridade, um fato que não ocorre entre os estados de um determinado *cluster* com os estados dos demais. A quantidade de 25 *clusters* foi adotada depois de vários testes em que os seguintes fatos foram observados: primeiro, para um número de *clusters* maiores que 25, foi notado a existência de *clusters* com um número muito pequeno de estados, onde em casos extremos, foram identificados *clusters* vazios. Por outro lado, quando o número de agrupamentos era significativamente menor que 25, foi notado que estados de tabuleiro muito distintos foram agrupados em um mesmo *cluster*. Devido a estes fatos, após uma série de testes empíricos foi concluído que 25 era um número de *clusters* que permitiria a obtenção de um processo de clusterização mais conciso.

O processo apresentado na Figura 46 é repetido para todos os estados existentes na base de dados. 80% dos estados de tabuleiro foram utilizados no processo de treinamento da Kohonen-SOM. Uma vez que a rede está treinada, cada estado desta base de dados é submetido à Kohonen-SOM para que fossem definitivamente associados ao *cluster* mais apropriado.

Após os estados de tabuleiro passarem pelo processo de clusterização, cada um dos *clusters* obtidos será utilizado no treinamento da MLP que o representa. A fim de manter o tempo de treinamento da rede neural razoável, o número de estados de tabuleiros de final de jogo em cada cluster também foi limitado a 25. A seleção dos estados que estarão presentes no *clusters* finais prioriza aqueles estados que ocorrem com maior frequência dentro daquele *cluster*. Cada MLP é treinada em torneio em que os jogos iniciam a partir de cada um dos estados de tabuleiro do referido *cluster*. Após treinada, cada MLP será especialista no perfil de tabuleiro de final de jogo representado por tal *cluster*.

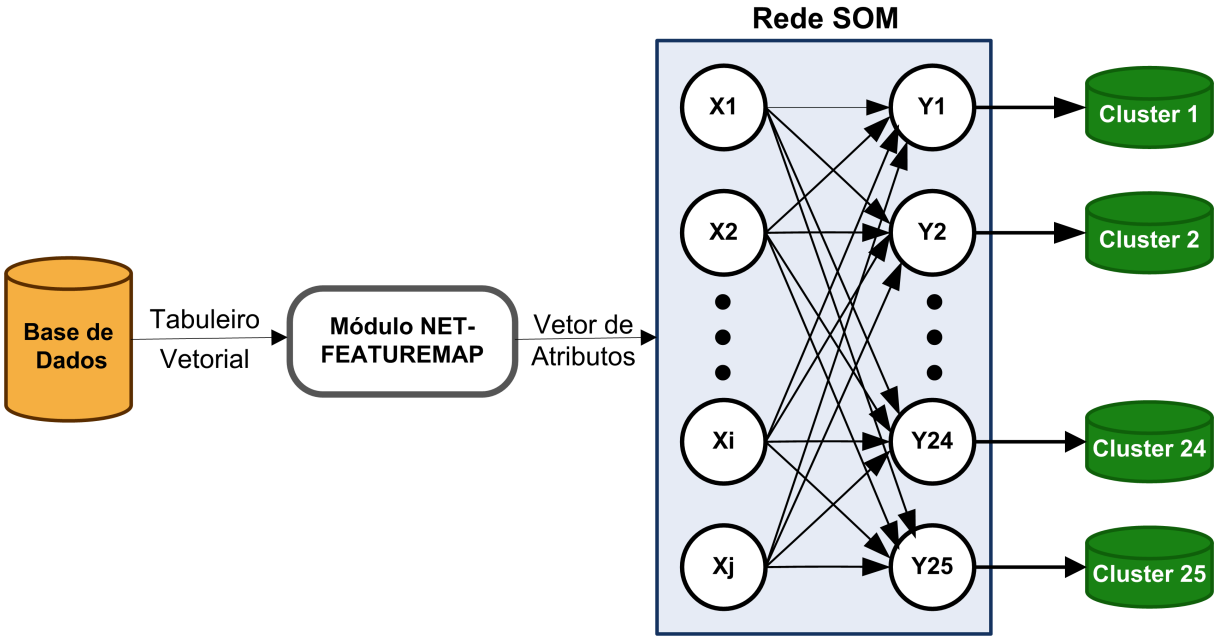


Figura 46 – Kohonen-SOM: processo de classificação da base de dados de final de jogo [3]

Uma vez que os agentes de final de jogo foram obtidos, o D-MA-Draughts está apto a disputar partidas. No entanto, este jogador pode optar entre duas dinâmicas de atuação em partidas. Tais dinâmicas são descritas na seção 7.3.

7.3 Dinâmicas de jogo do D-MA-Draughts

O D-MA-Draughts pode atuar em uma partida de disputa (sem reajuste de pesos da rede MLP) segundo duas dinâmicas. Tais dinâmicas se distinguem pela forma de cooperação entre os agentes de final de jogo. As seções 7.3.1 e 7.3.2 descrevem cada uma delas.

7.3.1 Dinâmica de Jogo I

A primeira dinâmica de jogo adotada pelo D-MA-Draughts, denominada DI, é ilustrada na Figura 47. O agente IIGA inicia a partida e a conduz até que o tabuleiro tenha, no mínimo, 13 peças. Quando o tabuleiro atinge no máximo 12 peças é caracterizado um estado de tabuleiro de final de jogo. A partir deste momento, este estado será enviado para a rede Kohonen-SOM que avaliará qual agente de final de jogo é o mais apto para se tornar o EGA da partida. Desta forma, uma vez definido o EGA este conduzirá a partida até o final.

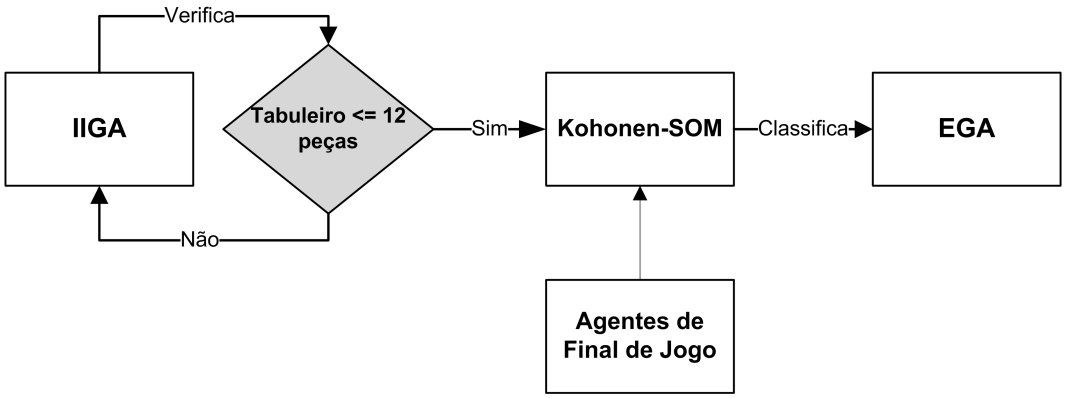


Figura 47 – Dinâmica de jogo I do D-MA-Draughts

7.3.2 Dinâmica de Jogo II

A segunda dinâmica de jogo adotada no D-MA-Draughts, denominada DII, é ilustrada na Figura 48. Como é possível observar, a DII consiste de uma pequena alteração na DI (seção 7.3.1), ou seja, o fluxo da partida é o mesmo até que seja atingido um estado de tabuleiro de final de jogo e um EGA seja definido. Desta forma, haverá uma interação na escolha do EGA, de modo que, a cada nova jogada a ser executada pelo D-MA-Draughts, a rede Kohonen-SOM verificará dentre os agentes de final de jogo o mais apto a se tornar o EGA e prosseguir a partida. Esta alteração foi realizada a fim de avaliar a cooperação de todos os agentes de final de jogo, visto que, após alguns movimentos, outro EGA poderá representar melhor o perfil do estado corrente do tabuleiro e, consequentemente, apontar a decisão mais acertada.

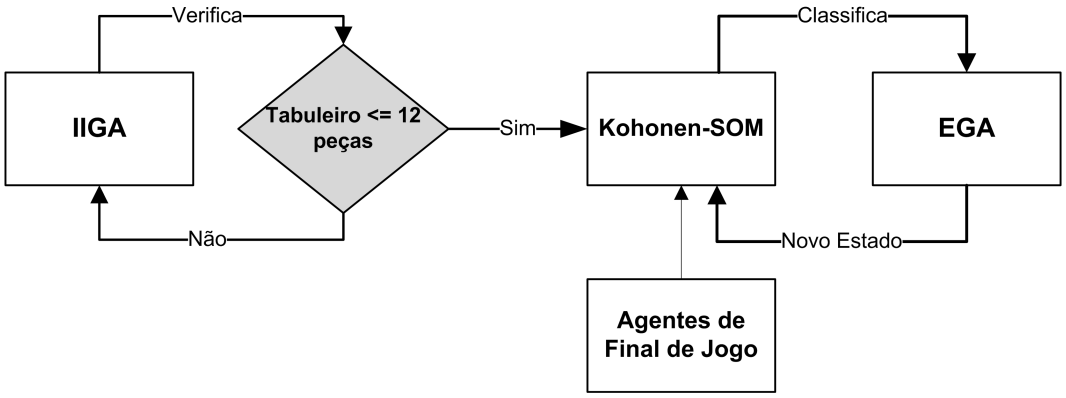


Figura 48 – Dinâmica de jogo II do D-MA-Draughts

Em Ref. [3], todas as análises experimentais foram realizadas considerando estas duas dinâmicas. Em tal trabalho foi concluído que a maior cooperação entre os agentes de final de jogo proporcionada pela DII, permitiu uma atuação mais eficaz nos torneios disputados. De fato, a segunda dinâmica apresentou melhor resultados na maioria das disputas realizadas entre o D-MA-Draughts e seus oponentes. Além disso, a DII também se destacou ao ser analisado os *loops* de final de jogo, uma vez que nos testes realizados houve

uma diminuição significativa deste problema. Ressalta-se que um *loop* ocorre quando o agente começa uma sequência repetitiva de movimentos (*loop*) alternando-se entre posições inúteis do tabuleiro, sendo que esses movimentos não modificam o estado do jogo [21]. Neste caso, o jogo tem que ser forçadamente finalizado após um determinado número de jogadas repetidas.

Apesar da boa performance da DII, em Ref. [3] não é apresentado um cenário de comparação entre a duas dinâmicas do D-MA-Draughts de modo a identificar em disputas entre elas qual apresenta melhor resultado. Uma análise desta natureza pode definir a dinâmica superior em relação a outra, e, consequentemente, definir a dinâmica que o D-MA-Draughts adotará permanentemente nas suas partidas.

7.4 Agentes Individuais do D-MA-Draughts

Cada agente do D-MA-Draughts consiste de uma rede MLP treinada por reforço através dos métodos das Diferenças Temporais TD (λ). Ressalta-se que, na primeira versão deste jogador, o mecanismo de tomada de decisão era composto pelo algoritmo distribuído YBWC (seção 3.1.1.3). Além disso, esta versão foi a utilizada na condução dos experimentos referentes à definição da dinâmica final a ser adotada em partidas, visto que, tal etapa foi cumprida logo no início do desenvolvimento deste trabalho de doutorado e as outras abordagens de distribuição do Alfa-Beta ainda estavam em período de estudo.

Sendo assim, a nova versão do D-MA-Draughts passa a contar com a arquitetura do ADABA-Draughts para cada um dos agentes, isto é, tanto para o IIGA, quanto para os agentes de final de jogo. Neste contexto, a arquitetura destes agentes pode ser visualizada no Capítulo 6. Neste contexto, as técnicas utilizadas no processo de aprendizado são as mesmas para todos os agentes do D-MA-Draughts, todavia, há algumas particularidades no treinamento do IIGA e dos agentes de final de jogo devido ao foco de cada um em uma partida. Estas particularidades são:

1. Como são inicializados os pesos da rede MLP;

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

No caso do agente IIGA, seu treinamento é conduzido a partir de um tabuleiro em estado inicial padrão de um jogo de Damas e os pesos da rede são iniciados aleatoriamente. O treinamento utiliza a técnica de *self-play* com clonagem, que, conforme apresentado na seção 6.3.3, realiza uma sequencia de jogos contra uma cópia do próprio jogador durante um determinado número de seções de forma a atualizar os pesos da rede neural MLP. Particularmente, o IIGA fez uso de 10 seções de 10 jogos.

Como os agentes de final de jogo tem o objetivo de atuar em fases finais da partida, mais precisamente, em estados de tabuleiro com 12 peças ou menos, não é conveniente

adotar o mesmo mecanismo de inicialização dos pesos da rede neural utilizado no IIGA. Além disso, o treinamento não é realizado a partir de um único estado de tabuleiro e sim a partir dos estados de tabuleiro *clusterizados* no processo descrito na seção 7.2. Todos os estados contidos nos *clusters* contém 12 peças. Por estes motivos, os pesos iniciais das redes MLP's dos agentes de final de jogo correspondem aos pesos do IIGA já treinado. A justificativa para este procedimento é que estes agentes não atuarão desde o início padrão de um jogo de Damas, logo a inicialização aleatória comprometeria a eficiência do jogador, visto que este se encontraria em desvantagem em relação a um adversário que se preparou para uma partida completa. Uma seção s de treinamento de um agente de final de jogo é composta por uma quantidade q de jogos e é realizada para cada um dos modelos de tabuleiro mt presentes no *cluster* i em questão. Cada *cluster* servirá de base para a criação de um agente de final de jogo. Desta forma, pode-se definir a quantidade de jogos de um processo de treinamento referente a um *agente* i como $s \cdot q \cdot mt$. Particularmente, no processo de treino de cada um dos agentes de final de jogo do *D-MA-Draughs* foram realizadas 3 seções compostas por 6 jogos para cada um dos 25 modelos de tabuleiros que compõem o *cluster* i, totalizando 450 jogos de treinamento para cada agente.

7.5 Considerações Finais

Este capítulo apresentou a arquitetura geral do SMA D-MA-Draughts. A construção deste agente foi realizada em dois momentos distintos. No primeiro momento, houve a definição da melhor dinâmica de atuação em partidas. Para isso, foi utilizada a primeira versão do D-MA-Draughts implementada pela autora deste trabalho em [3]. No segundo momento, cada um dos agentes do D-MA-Draughts passou a contar com a arquitetura do monoagente ADABA-Draughts. Desta forma, agora eles utilizam como mecanismo de tomada de decisão o algoritmo ADABA proposto neste trabalho. A avaliação do desempenho do D-MA-Draughts será apresentada na seção 8.4 do capítulo de experimentos.

Capítulo 8

Experimentos e Análise dos Resultados

Este capítulo apresenta os experimentos e análises dos resultados realizados no curso do desenvolvimento deste trabalho de doutorado. Os experimentos foram divididos em quatro etapas: comparação prática do desempenho das abordagens síncrona e assíncrona do Alfa-Beta (seção 8.1); avaliação do desempenho do algoritmo ADABA (seção 8.2); avaliação do monoagente jogador de Damas ADABA-Draughts (seção 8.3); e avaliação da plataforma multiagente jogadora de Damas D-MA-Draughts (seção 8.4). As considerações finais deste capítulo serão apresentadas na seção 8.5.

8.1 Comparação prática entre as abordagens de distribuição síncrona e assíncrona

Esta seção apresenta uma comparação prática entre as abordagens síncrona e assíncrona de distribuição do algoritmo de busca Alfa-Beta, representadas, respectivamente, pelas versões YBWC[33] e APHID[1]. Ressalta-se que no Capítulo 4 foi realizada uma comparação conceitual entre estas versões destacando suas particularidades em relação a abordagem de distribuição empregada. Portanto, complementando as observações apresentadas no Capítulo 4, ambas versões serão aplicadas a um mesmo problema prático de alta complexidade.

Neste contexto, o jogo de Damas foi utilizado como problema de aplicação, visto que possui uma complexidade considerável, conforme apresentado na Tabela 1. Para a condução destes experimentos foi utilizado o agente automático jogador de Damas D-VisionDraughts [19] implementado pela autora deste trabalho, que consiste de uma rede MLP que aprende pelos métodos das Diferenças Temporais (TD(λ)) e utiliza o algoritmo YBWC como mecanismo de tomada de decisão (detalhes em 3.2.2.4). Para o APHID, foi construído um agente automático jogador de Damas com arquitetura análoga à do D-VisionDraughts, mas com as tomadas de decisão conduzidas pelo algoritmo APHID, que foi integralmente implementado neste trabalho seguindo as diretrizes presentes em

[1].

Os experimentos foram executados em uma arquitetura de memória distribuída, que, sem dúvidas, configura o ambiente mais desafiador para qualquer proposta de distribuição do Alfa-Beta. Os parâmetros avaliados na comparação entre os algoritmos YBWC e APHID são: tempo de treinamento dos agentes; taxas de vitória; e avaliação da qualidade da escolha de movimentos segundo o "conselheiro" Cake - apresentado na seção 3.2.1.2. A arquitetura de memória distribuída utilizada é composta por 5 estações de igual configuração: processadores Qual Core - totalizando 20 núcleos de processamento - com 8GB de memória RAM. Todas estas estações foram interligadas por um *Switch Gigabit Ethernet*.

8.1.1 Tempo de Treinamento dos Agentes

As redes MLPs do D-VisionDraughts e do APHID-Draughts foram treinadas a partir da estratégia *self-play* com clonagem. Foram executados 100 jogos no processo de treino de cada MLP. Além disso, este processo foi realizado para cada uma das seguintes profundidades de busca: 10, 12 e 14. A Figura 49 ilustra o gráfico do tempo de treinamento obtido, em minutos, para os agentes APHID-Draughts e D-VisionDraughts em cada uma das profundidades de busca.

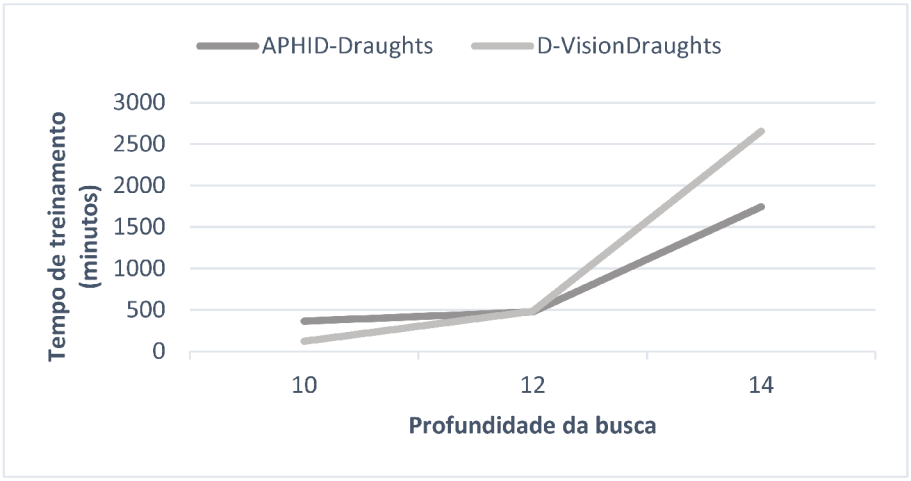


Figura 49 – Tempo de treinamento (em minutos) dos jogadores D-VisionDraughts e APHID-Draughts por profundidade

Como é possível observar, com profundidade igual a 10, o tempo de treino do D-VisionDraughts foi 66% (122 minutos) menor que o despendido pelo APHID-Draughts. No entanto, a medida que a profundidade de busca foi acrescida, este cenário foi invertido: o tempo de busca do APHID-Draughts na profundidade 12 (474 minutos) e 14 (1743 minutos) foi 1.6% e 35% menor, respectivamente, daqueles gastos pelo D-VisionDraughts (482 minutos para profundidade 12 e 2654 minutos para a profundidade 14). Estes resultados podem ser justificados pelos seguintes pontos:

1. O alto número de sincronizações necessárias no algoritmo YBWC resulta em processadores ociosos diversos períodos durante a busca. Este inconveniente é intensificado em profundidades maiores, pois quanto maior a profundidade de busca, maior o fator de ramificação, uma vez que estados avançados do tabuleiro de busca são encontrados, como por exemplo, aqueles que apresentam maior número de rainhas, fazendo com que existam mais opções de movimentos para o jogador.

2. O método de trabalho do YBWC, que é por roubo de trabalho (*working-stealing*), não é apropriado para arquitetura de memória distribuída, conforme explanado na seção 3.1.1.3.

3. O APHID-Draughts pode apresentar sobrecarga de busca acentuada, uma vez que não há compartilhamento de informações referentes a janela de busca. Como o YBWC tenta manter a janela de busca o mais próximo possível da versão serial devido ao compartilhamento dos limites alfa e beta, em profundidades mais rasas existe um equilíbrio favorável ao YBWC entre o número de podas e a sobrecarga de comunicação resultante das inúmeras mensagens inerentes ao YBWC. Neste ponto, é importante ressaltar que, a boa performance do APHID-Draughts se deve à adequada definição da largura da janela de busca dos escravos no APHID (conforme apresentado na seção 3.1.2.2). Neste trabalho, a abertura da janela foi definida empiricamente como 0.1.

Apesar desta seção ter considerado apenas o tempo de treinamento nas análises realizadas, é importante ressaltar que, apesar de cada jogador ter a mesma arquitetura e ter executado o treinamento de suas respectivas MLPs em profundidades iguais, as MLPs treinadas resultante de cada jogador são distintas. Tal fato ocorre, pois apesar de o APHID e o YBWC serem versões paralelas do algoritmo Alfa-Beta, o resultado final do algoritmo pode não ser idêntico para um mesmo estado de jogo em função da definição dos limites da janela de busca no APHID, pois a solução ótima (que seria retornada pelo algoritmo serial) pode ficar fora do intervalo de busca.

8.1.2 Torneios

A Tabela 5 apresenta as taxas de vitória obtidas pelo APHID-Draughts em um torneio em que seu oponente foi o D-VisionDraughts. Cada jogador utilizou sua respectiva MLP treinada na profundidade 14. A estratégia de busca adotada foi por ID, sendo que cada jogada foi limitada a 30 segundos. Os jogos contemplaram 20 estados diferentes, onde cada estado tinha o mesmo número de peças para cada jogador (preto ou vermelho). Este torneio foi executado em duas rodadas: na primeira, o jogador APHID-Draughts jogou com peças pretas e o D-VisionDraughts com as peças vermelhas; e, na segunda, a situação foi invertida, isto é, o APHID-Draughts jogou com peças vermelhas e o D-VisionDraughts

com peças pretas. Desta forma, o torneio contou com o total de 40 jogos. Assim, não houve favorecimento de nenhum jogador, uma vez que poderia haver estados de tabuleiro que beneficiassem quem estivesse jogando com as peças vermelhas e vice-versa.

Tabela 5 – Taxas de vitória obtidas pelo APHID-Draughts

Rodada	**Vitórias**	**Empates**	**Derrotas**
Rodada I	75%	20%	5%
Rodada II	55%	40%	5%

A partir dos resultados apresentados na Tabela 5 é possível observar que o APHID-Draughts jogando com peças pretas apresentou uma taxa de 75% de vitórias contra 5% do seu oponente, o D-VisionDraughts. Tal fato corresponde a uma taxa de 70% de superioridade do APHID-Draughts em relação ao seu oponente. A taxa de superioridade é calculada a partir da diferença entre as porcentagens de vitórias obtidas por cada jogador. Ao considerar o APHID-Draughts jogando com peças vermelhas, ele apresentou uma taxa de 50% de superioridade em relação ao D-VisionDraughts.

Neste contexto, é possível concluir que o APHID-Draughts se sobressaiu jogando tanto com peças pretas quanto com peças vermelhas. A fim de comprovar esta vantagem do APHID-Draughts em relação ao D-VisionDraughts, foi empregado sobre estes resultados o método estatístico não-paramétrico *Wilcoxon signed rank test*, conforme detalhado na seção 8.1.2.1 a seguir.

8.1.2.1 Análise 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, estes valores correspondem a um valor médio que é obtido através de várias execuções [77]. 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 do teste de *Wilcoxon* é comparar o desempenho de cada algoritmo (ou pares de algoritmos, por exemplo A e B) no sentido de verificar se existem diferenças significativas entre os seus resultados (amostras ou populações).

Resumidamente, o método 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 ($-$). Estas 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 po-

sitivos (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 deste segundo grupo, dá-lhe o valor de uma estatística designada por *p-value*, que pode ser consultada na tabela de significância apropriada [77]. 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). Caso se verifique 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. Mais detalhes sobre o método estatístico de Wilcoxon podem ser obtidos em [77].

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 capítulo, APHID-Draughts, apresenta ou não um desempenho diferenciado em relação ao seu adversário D-VisionDraughts. Então, nas análises dos resultados consolidados na Tabela 5, o método de Wilcoxon assume, como ponto de partida, a seguinte hipótese nula H_0: APHID-Draughts possui o mesmo nível de desempenho que o D-VisionDraughts.

De acordo com o método de Wilcoxon, os seguintes passos devem ser seguidos a fim de provar que H_0 deve ser rejeitada [18]:

Passo 1: Dado T como a menor soma dos *ranks* (ordenações) positivas e negativas dentre as diferenças dos pares de amostras, isto é, $T = Min\{R^+, R^-\}$ (particularmente neste trabalho, R^+ representa a soma dos *ranks* em que o APHID-Draughts superou o D-VisionDraughts, enquanto R^- é a soma dos *ranks* em que o APHID-Draughts foi superado pelo D-VisionDraughts). Uma apropriada tabela estatística ou ferramenta deve ser utilizada para determinar o *teste estatístico*, *valor crítico* ou *p-value* (dado provido pelo teste);

Passo 2: Rejeita-se a *hipótese nula* H_0 se o *teste estatístico* $\leq$ *valor crítico* ou se *p-value* $\leq \alpha$ (nível de significância). Quanto mais baixo é o valor de α, mais forte é a evidência contra a *hipótese nula* H_0.

A Tabela 6 apresenta os testes estatísticos de *Wilcoxon* aplicados sobre os resultados dos jogos executados na seção 8.1.2, onde a seguinte pontuação 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 6 – Método *Wilcoxon signed rank test* aplicado aos resultados do torneio entre o APHID-Draughts e D-VisionDraughts

Torneio	R^+	R^-	***p-value***
APHID-Draughts x D-VisionDraughts	377	29	0,000006

Como é possível observar, o APHID-Draughts se mostrou significativamente superior ao D-VisionDraughts com um alto nível de significância $\alpha = 0.01$ ou 99%. Tal fato representa que há 1% de chance da *hipótese nula* H_0 (equivalência de desempenho entre o APHID-Draughts e o D-VisionDraughts) ocorrer na base de dados (amostra) analisada. Em outras palavras, os dados apresentados na Tabela 6 rejeitam fortemente H_0 o que significa que os resultados alcançados pelo APHID-Draughts, nos jogos de torneio apresentados na Tabela 5, são de fato superiores em relação ao D-VisionDraughts.

A boa performance do algoritmo APHID-Draughts é justificada por suas características que não o limitam em uma arquitetura de memória distribuída e pela definição adequada da janela de busca, que permite controlar a sobrecarga de busca e, consequentemente, melhora o *look-ahead* do jogador quando os jogos são executados em ID.

8.1.3 Avaliação da Qualidade dos Movimentos em Relação ao Cake

Os experimentos realizados nesta subseção tem como proposta a avaliação do nível da qualidade do processo de tomada de decisão dos jogadores APHID-Draughts e D-VisionDraughts. Esta avaliação é baseada na taxa de coincidência entre os movimentos escolhidos por cada agente e os movimentos que o agente Cake poderia realizar nas mesmas situações. Existem duas razões para a utilização do Cake como "conselheiro" de movimentos: primeiro, ele é um jogador de alto nível; segundo, o fato de que ele está disponível na plataforma CheckerBoard conforme introduzido na seção 3.2.1.2. A fim de lidar com este objetivo, as informações de movimento foram obtidas a partir do mesmo torneio apresentado na seção 8.1.2 e registradas em um arquivo do tipo PDN (*Portable Draughts Notation*). Este formato de arquivo é o padrão para os jogos de Damas [63]. Além disso, em Ref. [78] foi implementada uma interface que torna possível a comparação automática das escolhas dos movimentos realizados por agentes jogadores de Damas como o APHID-Draughts e o D-VisionDraughts com o Cake por meio da plataforma CheckerBoard. Ressalta-se que o nível de jogo bastante elevado do Cake - que o torna uma referência indiscutível para ser usado como lastro de comparação - foi atingido através de um processo fortemente supervisionado de aprendizagem [78]. Neste sentido, utilizar a comparação com o Cake como métrica de qualidade não representa a intenção de tornar

o agente jogador avaliado uma cópia do mesmo. Na verdade, parte-se do princípio de que, como o Cake sempre retorna a melhor opção para um determinado movimento dado o seu alto nível de jogo, considerar que o agente avaliado realizou diversas tomadas de decisão equivalentes às que o Cake tomaria na mesma situação, permite concluir que os movimentos do agente avaliado possuem uma boa qualidade.

A Tabela 7 mostra os resultados comparativos nas situações em que ambos os agentes operaram como jogador preto (primeira linha da Tabela) e como jogador vermelho (segunda linha). Coerentemente com os resultados anteriores, estes resultados confirmam a superioridade do APHID-Draughts em relação ao D-VisionDraughts. De fato, o APHID-Draughts jogando com peças pretas obteve 66.53% contra 30,03% do D-VisionDraughts e, jogando com peças vermelhas, apresentou uma taxa de coincidência de 43,9% contra 33,89% do seu oponente.

Tabela 7 – Taxas de coincidência de movimentos com o Cake

Posição do agente no jogo	APHID-Draughts	D-VisionDraughts
Peças Pretas	66.53%	30.03%
Peças Vermelhas	43.9%	33.89%

8.2 Avaliação do desempenho do algoritmo ADABA

Esta seção apresenta os experimentos realizados para avaliar o desempenho do algoritmo ADABA cujos detalhes da implementação foram descritos no Capítulo 5. Para isso, primeiramente, a seção 8.2.1 apresenta a metodologia empregada e, na sequência, serão avaliados os seguintes parâmetros: variação do tempo de tomada de decisão (seção 8.2.2); *Speed-Up* (seção 8.2.3); eficiência (seção 8.2.4); sobrecarga de busca (seção 8.2.5); e qualidade da solução (seção 8.2.6).

8.2.1 Metodologia

Definição dos algoritmos avaliados

A avaliação do desempenho do ADABA foi realizada comparando-o com a versão assíncrona APHID e com o Alfa-Beta serial. Com o intuito de mostrar a evolução do ADABA por meio das contribuições propostas nesta tese de doutorado, este algoritmo foi avaliado considerando três versões distintas:

- **ADABA-V1:** Nesta versão considerou-se apenas a contribuição referente à automatização do processo de definição e atualização das janelas de busca nos processadores escravos apresentada na seção 5.3.

- ❑ **ADABA-V2:** Nesta versão considerou-se a contribuição que se refere a inclusão de uma nova política de prioridades para a seleção das tarefas que os escravos vão selecionar para exploração (seção 5.1.2). Além desta, também foi considerada a contribuição que se refere ao novo mecanismo utilizado pelo processador mestre para o tratamento das tarefas recebidas dos escravos apresentado na seção 5.4.

- ❑ **ADABA-V3:** Esta versão consiste na junção das versões V1 e V2. É importante ressaltar que esta é a versão final do algoritmo proposto neste trabalho de doutorado.

No caso do APHID, foi utilizada a mesma implementação empregada nos testes apresentados na seção 8.1. No entanto, a fim de observar o comportamento do APHID devido à necessidade da definição empírica e manual da constante de formação da janela de busca nos processadores escravos - fato que foi descrito como uma fragilidade na seção 4.3 - foram consideradas 3 versões que variaram entre si apenas em relação ao valor da constante C que delimita a largura da janela alfa-beta, conforme apresentado na seção 3.1.2.2. Neste contexto, as referidas versões são:

APHID-V1: $C = 0.1$, aqui este valor equivale ao utilizado nos experimentos do APHID-Draughts apresentados na seção 8.1;

APHID-V2: $C = 0.5$ a fim de observar uma situação cujos limites da janela de busca estabeleçam um intervalo maior do que o utilizado nos testes da seção 8.1;

APHID-V3: $C = 0.01$ para compor uma situação de maior estreitamento da janela de busca comparada à janela utilizada nos testes da seção 8.1.

O Domínio e Configuração do Problema

O domínio do jogo de Damas foi usado como problema de aplicação. Desta forma, a arquitetura do agente jogador ADABA-Draughts descrita na seção 6.1 foi utilizada como base para a construção das três versões que empregam o ADABA e também daquelas que usam o APHID. Por simplificação, na descrição dos resultados será utilizado apenas o nome do algoritmo e sua referida versão.

Na arquitetura do ADABA-Draughts, o valor da solução da busca é obtido pela expansão da árvore de jogo, onde os nós folhas são submetidos à avaliação da MLP que retorna a predição equivalente ao estado de tabuleiro avaliado. A fim de focar apenas na avaliação dos algoritmos, todos os agentes tratados nestes experimentos fizeram uso da mesma rede MLP que corresponde àquela utilizada nos testes da seção 8.1.1. Assim, para um mesmo estado, existe a possibilidade do valor da avaliação ser idêntico.

Os estados de tabuleiro utilizados foram extraídos da base de testes *Tinsley-Chinook Test Suite 1992* [79] e podem ser visualizados no Apêndice A. Esta base contem os estados dos jogos disputados entre o campeão mundial em Damas da época, Dr. Marion Tinsley,

e o jogador automático de Damas Chinook em um torneio de 40 jogos [80]. Desta base, foram selecionados 15 estados cujo o próximo movimento deve ser indicado pelo jogador preto. A seleção destes estados se baseou nos mesmos estados utilizados por Brockington nos testes do APHID reportados em [1]. Além destes estados, foi incluído o estado inicial padrão do jogo de Damas. Portanto, os experimentos foram executados sobre um conjunto de 16 estados de tabuleiro.

Cada versão dos algoritmos analisados foi executada isoladamente para cada estado. Desta forma, foi possível comparar as versões dos algoritmos nas mesmas situações. Tal medida foi adotada, pois como as versões distribuídas podem divergir em relação ao valor minimax retornado como solução da busca, se fosse considerado o conjunto de estados obtidos no percurso de um jogo, poderia haver uma superestimação da velocidade de uma versão distribuída avaliada, uma vez que cada agente poderia explorar estados diferentes ao longo do jogo. Neste caso, é importante perceber que determinados estados geram uma árvore de busca muito pequena e outros uma árvore de busca muito maior. Ainda com a intenção de criar um cenário de igualdade para avaliar os algoritmos, todas as execuções foram realizadas utilizando a profundidade de busca igual a 12.

Hardware

Os testes foram executados a partir de 4 estações de igual configuração: processadores Quad Core - totalizando 16 núcleos de processamento - com 8GB de memória RAM. Estas estações foram interligadas por um *Switch Gigabit Ethernet*.

Para a execução dos experimentos cada estação executou até 4 processos, o que permitiu atingir o valor máximo de 16 processadores executando simultaneamente. Neste sentido, diversas configurações foram definidas variando o número de processadores utilizados ao longo das execuções. Estas configurações foram definidas conforme a Tabela 8.

Tabela 8 – Configurações das estações de trabalho em relação ao número de processadores utilizados nos experimentos

Número de Processadores do Experimento	**Número de Processos Alocados**			
	Estação 1	Estação 2	Estação 3	Estação 4
2	1	1	0	0
4	1	1	1	1
7	1	2	2	2
10	1	3	3	3
13	1	4	4	4
16	4	4	4	4

Software

Tanto o APHID quanto o ADABA seguem o modelo de paralelismo mestre-escravo. Desta forma, as instruções inerentes ao processador mestre são distintas daquelas próprias de um processador escravo. Neste contexto, para cada algoritmo foi compilado dois módulos: um referente ao processador mestre e outro referente ao processador escravo. Neste sentido, apenas uma das máquinas (Estação 1) recebeu ambos os módulos, visto que em algumas configurações dos experimentos ela executará ambos os papéis (mestre e escravo).

Linguagem de Programação

Os algoritmos foram implementados na linguagem de programação Java e executados em uma JVM versão 1.8 rodando sobre o Sistema Operacional Ubuntu versão 14.04. Nas versões distribuídas, para comunicação entre os processos foi utilizada uma interface de comunicação remota através da biblioteca RMI (*Remote Method Invocation*) [81].

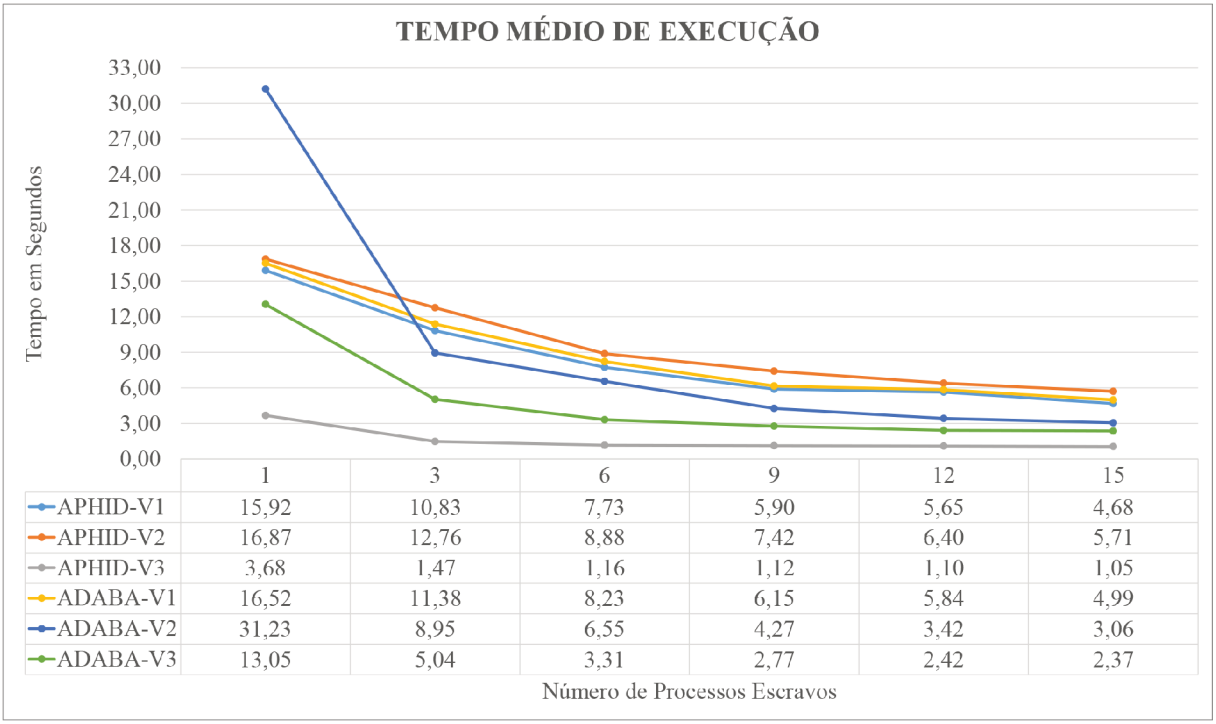
8.2.2 Variação do tempo de tomada de decisão

Para verificar o tempo médio de tomada de decisão, todos os algoritmos (incluindo suas versões) foram executados 6 vezes. Em cada execução foi adotado um determinado número de processadores escravos conforme apresentado na Tabela 8. Assim, a primeira execução foi conduzida com um processador mestre e apenas um escravo. As execuções subsequentes foram executadas com um processador mestre, mas com um incremento de 3 processadores escravos até o limite da arquitetura de hardware disponível, isto é, até 15 processadores escravos.

O gráfico da Figura 50 mostra o tempo médio gasto na execução dos algoritmos para o conjunto de estados de tabuleiro variando o número de processadores escravos. Assim, observando-se o gráfico é possível notar os seguintes pontos comuns entre todas as versões:

1. o tempo médio de execução é inversamente proporcional ao número de processadores escravos utilizados. Esta situação ocorre, pois a medida que processadores são adicionados, existe uma maior divisão de tarefas que serão executadas em paralelo, o que auxilia na agilidade da busca.

2. o maior tempo de processamento ocorreu com a utilização de apenas um processador escravo. Tal comportamento é justificado pelo fato de que uma versão distribuída do Alfa-Beta possui diversas instruções adicionais o que torna o algoritmo mais complexo. Além disso, ainda existe o tempo gasto com trocas de mensagens e atualizações de dados nas estruturas de dados utilizadas.



	1	3	6	9	12	15
APHID-V1	15,92	10,83	7,73	5,90	5,65	4,68
APHID-V2	16,87	12,76	8,88	7,42	6,40	5,71
APHID-V3	3,68	1,47	1,16	1,12	1,10	1,05
ADABA-V1	16,52	11,38	8,23	6,15	5,84	4,99
ADABA-V2	31,23	8,95	6,55	4,27	3,42	3,06
ADABA-V3	13,05	5,04	3,31	2,77	2,42	2,37

Figura 50 – Tempo médio de tomada de decisão dos algoritmos distribuídos avaliados com variação do número de processadores escravos

3. O decremento do tempo médio de execução foi mais acentuado até a utilização de 6 processadores escravos. A partir de então, apesar de haver uma diminuição do tempo de execução, a redução foi mais suave. Esta situação está relacionada ao fato de que quanto mais processos envolvidos na busca, maior a sobrecarga de comunicação, pois existem mais pacotes gerados pelos escravos sendo enviados ao mestre ao mesmo tempo.

Em uma visão direcionada para cada uma das versões, é possível notar que o APHID-V3 apresentou o menor tempo médio de execução independentemente do número de processadores escravos. Esta situação justifica-se por esta versão ter utilizado um valor muito pequeno para formação dos limites da janela de busca, conforme descrito na seção 4.3. Desta forma, com uma janela estreita, o número de podas foi muito elevado culminando na celeridade do processo de busca. No entanto, convém salientar que a utilização de uma janela de busca muito estreita pode comprometer a qualidade da busca, uma vez que a solução ótima pode ser perdida. Este aspecto será comentado com mais detalhes na seção 8.2.6 que analisará a qualidade da solução retornada pelos algoritmos. Por outro lado, nota-se que a versão do APHID-V2 foi, de modo geral, mais lenta que as demais versões. A única exceção ocorreu na execução do ADABA-V2 com apenas um processador escravo. O tempo de execução mais elevado do APHID-V2 também relaciona-se com o valor definido para a constante de formação da janela de busca. Todavia, neste caso, a constante

permitiu uma abertura maior entre os limites alfa e beta. Assim, diversas podas foram perdidas durante o processo de busca e, consequentemente, diversos nós foram explorados sem necessidade agravando a ocorrência de sobrecarga de busca, conforme será explanado na seção 8.2.5.

No caso do ADABA-V2, realmente houve um tempo muito elevado quando ele foi executado com um processador escravo, mas com 3 ou mais processadores escravos ele tornou-se mais rápido que o APHID-V1, APHID-V2 e ADABA-V1. A razão deste comportamento está relacionado à forma como esta versão opera: ela utiliza um *pool de threads* para tratar as mensagens recebidas dos processadores escravos (detalhes na seção 5.4). O problema é que quando há apenas um processador escravo, todos os pacotes recebidos dos escravos devem atualizar a mesma porção da estrutura de dados AIS do mestre. Neste caso, as *threads* entram em "condição de corrida" (*race condition*)[1] [82]. Evidentemente, este comportamento do ADABA-V2 seria eliminado se fosse considerada apenas uma *thread* no tratamento dos dados recebidos. Todavia, a mesma configuração foi mantida para todas as execuções para permitir verificar o comportamento geral desta versão do algoritmo. Quando são incluídos mais processadores escravos, é possível observar a existência de um melhor aproveitamento do processamento paralelo provido pelas *threads*, uma vez que existe mais chances de processadores distintos enviarem pacotes simultaneamente. Como estes pacotes serão atualizados em *AIS* distintas, o problema de condição de corrida deixa de apresentar impacto negativo, conforme pode ser observado nos resultados obtidos pelo ADABA-V2.

As versões APHID-V1 e ADABA-V1 apresentaram valores muito próximos no tempo de execução em todas as variações do número de processadores. No entanto, nota-se que o APHID-V1 apresentou melhor tempo que o ADABA-V1 em todos os casos. Salienta-se que o APHID-V1 utiliza um valor que foi considerado satisfatório para a definição da largura da janela de busca para o domínio do problema utilizado no contexto deste trabalho, conforme explanado nos experimentos da seção 8.1. Todavia, este valor foi obtido de modo empírico e manual. O ADABA-V1 implementa uma nova política para definição automática da largura da janela de busca . Neste sentido, apesar do tempo de execução dele ser maior que o APHID-V1, é possível observar que a política proposta pelo ADABA oferece um tempo de execução próximo ao da melhor versão obtida do APHID para o problema tratado.

O ADABA-V3 apresentou um tempo médio de execução muito bom, ele apenas foi mais lento do que o APHID-V3. Este resultado demonstra que as contribuições deste trabalho, em conjunto, permitiram a obtenção de um tempo médio de execução superior às demais versões. A fim de expandir as análises referentes ao tempo de execução dos algoritmos, as seções 8.2.3 e 8.2.4 apresentam, respectivamente, as observações sobre o

[1] A "condição de corrida" (*race condidtion*) ocorre quando duas ou mais *threads* tentam acessar o mesmo recurso (arquivo, memória, etc.) simultaneamente [82].

Speed-Up e eficiência observados nas versões dos algoritmos avaliados.

8.2.3 *Speed-Up*

Esta seção avalia a aceleração (*speed-up*) dos algoritmos, que, assim como apresentado na seção 2.4.1, é calculada pela razão entre o tempo de execução da versão serial e paralela. A Tabela 9 apresenta os valores obtidos do *speed-up* apresentado por cada uma das versões avaliadas em função da quantidade de processadores escravos. Os valores apresentados nesta tabela estão vinculados com os tempos de execução obtidos nos testes da seção 8.2.2.

Ao analisar a Tabela 9, observa-se que as versões APHID-V1, APHID-V2, ADABA-V1 e ADABA-V2 apresentaram *speed-up* positivo (velocidade superior à da versão serial) a partir da utilização de 9 processadores escravos, ou seja, nas situações em que foram utilizados menos processadores estas versões foram mais lentas do que a versão serial do Alfa-Beta ou apresentaram um tempo de execução muito próximo ao dele. O ADABA-V3 mostrou-se mais rápido que a versão serial a partir da utilização de 3 processadores escravos. Apenas a versão APHID-V3 teve aceleração superior àquela praticada pela versão serial do Alfa-Beta independente do número de processadores escravos envolvidos. Estes resultados corroboram com o fato de que uma versão distribuída é mais complexa do que a sua respectiva versão serial. Além disso, é necessário considerar que a utilização de arquitetura de memória distribuída contribui para o aumento do tráfego de mensagens gerando sobrecarga de comunicação (seção 2.4.2). Nesta direção, o emprego da distribuição da busca torna-se viável quando:

1. há um número adequado de processadores escravos. Entretanto, um ponto a se destacar é que o incremento do número de processadores escravos possui um limite. A partir deste limite começa a "desaceleração" do processo da busca. Esta situação ocorre devido ao aumento da sobrecarga de comunicação. Infelizmente, devido à limitação de arquitetura, não foi possível concluir neste trabalho a partir de quantos processadores escravos o processo de busca começaria a sofrer uma degradação de desempenho.

2. deseja-se explorar níveis mais profundos da árvore de busca. Neste caso, a versão serial torna-se tão lenta que é impraticável a sua execução.

8.2.4 Eficiência

O cálculo da eficiência dos algoritmos está diretamente relacionado com o cálculo do *speed-up*, uma vez que o seu valor é dado pela razão entre o *speed-up* e o número de processadores utilizados (seção 2.4.1). Salienta-se que tanto o APHID quanto o ADABA

Tabela 9 – *Speed-Up* observado para cada algoritmo variando o número de processadores escravos

Algoritmo	**Número de Processadores**					
	1	3	6	9	12	15
APHID-V1	0,47	0,70	0,98	1,28	1,34	1,62
APHID-V2	0,45	0,59	0,85	1,02	1,18	1,32
APHID-V3	2,06	5,14	6,50	6,73	6,90	7,21
ADABA-V1	0,46	0,66	0,92	1,23	1,29	1,51
ADABA-V2	0,24	0,84	1,00	1,77	2.21	2,47
ADABA-V3	0,58	1,50	2,28	2,73	3,12	3,19

na situação mais simples utilizará dois processadores: 1 mestre e 1 escravo. Por esta razão, no cálculo da eficiência é necessário considerar o processador mestre também. A Tabela 10 apresenta a eficiência obtida por cada uma das versões dos algoritmos em função da quantidade de processadores. Foram apresentados apenas os valores das situações em que o *speed-up* foi superior a 1 na Tabela 9.

Tabela 10 – Eficiência apresentada por cada algoritmo variando o número de processadores

Algoritmo	**Número de Processadores**					
	2	4	7	10	13	16
APHID-V1				0,13	0,10	0,10
APHID-V2				0,10	0,09	0,08
APHID-V3	1,03	1,28	0,93	0,67	0,53	0,45
ADABA-V1				0,12	0,10	0,09
ADABA-V2				0,18	0,17	0,15
ADABA-V3		0,37	0,33	0,27	0,24	0,20

Na Tabela 10 nota-se que há um comportamento inverso ao da aceleração, isto é, a medida que novos processadores são adicionados à busca, menor a eficiência dos algoritmos. Tal fato ocorre, pois quanto maior o número de processadores, maior a sobrecarga de comunicação, o que contribui para o atraso das execuções. Além disso, é importante salientar que nos testes cada estação de trabalho contém 4 núcleos de processamento. Neste contexto, quando há mais de um processo executando na mesma estação deve-se considerar os seguintes fatores de limitação: 1) há um compartilhamento da memória RAM; 2) Por se tratar de uma estação de trabalho convencional, existem diversos outros processos inerentes ao Sistema Operacional disputando estes núcleos de processamento. Um cuidado tomado no contexto da configuração dos experimentos (seção 8.2.1) foi limitar o número de processos em cada estação de trabalho à quantidade de núcleos existentes. A intenção foi usufruir dos benefícios da arquitetura *multicore*, sem agravar os fatores de

limitação relacionados anteriormente. Acredita-se que utilizar mais estações de trabalho executando um número de processos inferior ao limite do total de núcleos existentes aumentaria o desempenho dos algoritmos. Um fato que contribui para essa crença é a situação do ADABA-V3 que atingiu uma eficiência de 0.37 quando utilizou um processo escravo em cada estação de trabalho, conforme configuração apresentada na Tabela 8.

Apesar da eficiência dos algoritmos não estarem próximas da ideal, isto é, 1, deve-se atentar para os benefícios que a aceleração trás para um ambiente de elevada complexidade do espaço de estados. Quanto mais rápido o algoritmo é, mais estados ele poderá explorar em um determinado intervalo de tempo. Desta forma, as versões distribuídas permitem um melhor desempenho da busca justamente pelo fato de permitir explorar mais profundamente a árvore de busca (*look-ahead*).

Considerando a versão do APHID-V1 (que é a versão oficial para o APHID-Draughts, seção 8.1) e o ADABA-V3 (versão final do ADABA), este último foi, pelo menos, duas vezes mais rápido que o primeiro. Na prática, essa eficiência traz os seguintes benefícios:

- Se for fixada uma profundidade de busca, a solução será retornada mais rapidamente. No caso particular do treinamento de uma rede MLP, como é o caso da arquitetura do agente jogador utilizado neste trabalho, esta agilidade é positiva visto que o processo como um todo é moroso (mais detalhes do treinamento da rede MLP da arquitetura do agente jogador de Damas tratados neste trabalho podem ser visualizados na seção 6.3).

- Se for fixado um limite de tempo para as jogadas, a versão mais ágil atingirá profundidades da árvore de busca superiores. Neste caso, se for considerado o período de treinamento de uma rede MLP, certamente a qualidade da mesma será melhor. Por outro lado, se for um jogo de disputa, a tomada de decisão será mais acurada.

Ainda observando a Tabela 10, destaca-se a eficiência apresentada pelo APHID-V3 que com 2 e 3 processadores chegou a apresentar valores superiores à eficiência ideal. Tal fato demonstra que o APHID pode ser muito eficiente quando é definido um valor pequeno de abertura da janela de busca. Todavia, esta aceleração pode degradar a qualidade da busca, conforme será apresentado na seção 8.2.6.

8.2.5 Sobrecarga de busca

A sobrecarga de busca foi calculada pela diferença do total de nós explorados pelas versões distribuídas e o total explorado pela versão serial. Portanto, ela representa o número de nós que a versão distribuída explorou a mais do que a serial na mesma situação. A Tabela 11 apresenta a porcentagem de sobrecarga de busca apresentada por cada versão dos algoritmos nas diversas execuções em que houve a variação do número de processadores escravos.

Tabela 11 – Porcentagem da sobrecarga de busca apresentada por cada versão dos algoritmos avaliados.

Algoritmo	**Número de Processadores Escravos**					
	1	**3**	**6**	**9**	**12**	**15**
APHID-V1	51,64%	73,48%	76,10%	75,40%	76,29%	68,55%
APHID-V2	54,37%	77,41%	79,93%	80,80%	80,32%	75,55%
APHID-V3	0	0	0	0	0	0
ADABA-V1	56,89%	74,62%	77,10%	76,19%	76,65%	67,08%
ADABA-V2	75,45%	72,15%	72,98%	72,48%	70,52%	60,07%
ADABA-V3	50,02%	39,23%	40,51%	32,60%	38,17%	30,74%

Ao observar a Tabela, nota-se que os resultados seguem o padrão observado na seção 8.2.2, uma vez que a sobrecarga de busca interfere muito no tempo de execução dos algoritmos, pelos seguintes motivos:

1. Se há mais nós a serem explorados, há mais tempo de processamento sendo dispendido;

2. O atraso provocado pelo motivo 1 faz com que o processador mestre efetue uma varredura com informações menos precisas, o que pode acarretar em mais nós na borda da busca que deverão ser distribuídos aos escravos;

3. Com mais nós sendo distribuídos, há mais informação trafegando pela rede, o que contribui para a sobrecarga de comunicação.

A sobrecarga de busca está diretamente relacionada com a definição da janela de busca. De fato, se os valores dos limites formam uma janela de busca muito estreita, há um número muito alto de podas, o que permite que seja explorado menos nós do que a versão serial exploraria na mesma situação. Este caso é claramente visualizado no APHID-V3, que não apresentou sobrecarga em situação alguma. Por outro lado, se os limites alfa e beta constituem uma janela de busca mais larga, um número maior de nós são explorados sem necessidade, assim como ocorreu com o APHID-V2 que chegou a explorar até 80% mais nós do que a versão serial nas execuções com 9 e 12 processadores escravos.

A Tabela 11 também permite notar que as versões APHID-V1 e ADABA-V1 apresentaram valores de sobrecarga de busca muito próximos em todas as execuções. Este comportamento sugere que a política de formação automática da janela de busca representada pelo ADABA-V1 permitiu a criação de uma janela de busca próxima daquela utilizada pela melhor versão obtida do APHID para o problema tratado (representada pelo APHID-V1).

Ao observar o ADABA-V2 nota-se que na execução com um processador escravo, a sobrecarga de busca foi muito elevada, maior inclusive do que a apresentada pelo APHID-V2. Esta situação é consequência do mesmo problema de "condição de corrida" explanado na seção 8.2.2. No entanto, com mais processadores escravos, há a diminuição da sobrecarga de busca, sendo que o ADABA-V2 passou a apresentar taxas inferiores ao APHID-V1 e ADABA-V1. Por exemplo, com 15 processadores escravos o ADABA-V2 teve uma redução de 8,48% em relação ao APHID-V1.

O ADABA-V3, dentre as versões que apresentaram sobrecarga de busca, obteve a menor taxa em todas as execuções. Tal resultado se deve às estratégias utilizadas nesta versão que é a junção das contribuições representadas pelo ADABA-V1 e ADABA-V2.

8.2.6 Qualidade da Solução

A fim de avaliar a qualidade da solução dos algoritmos, esta seção faz uma análise da taxa de coincidência do valor retornado por cada uma das versões avaliadas com aquele que a versão serial retornaria na mesma situação. Salienta-se que a versão serial retorna a resposta ótima. Nesta direção, a Tabela 12 apresenta as porcentagens de coincidência dos algoritmos com a versão serial.

Tabela 12 – Taxa de coincidência com a versão serial do Alfa-Beta

Algoritmo	**Número de Processadores Escravos**					
	1	3	6	9	12	15
APHID-V1	43,75%	50,00%	75,00%	68,75%	68,5%	68,75%
APHID-V2	43,75%	68,75%	81,25%	81,25%	81,25%	81,25%
APHID-V3	6,25%	0	6,25%	6,25%	6,25%	6,25%
ADABA-V1	43,75%	37,50%	43,75%	43,75%	43,75%	50%
ADABA-V2	68,75%	75,00%	81,25%	75,00%	75,00%	75,00%
ADABA-V3	62,50%	56,25%	62,50%	68,75%	62,50%	62,50%

Na Tabela 12, observa-se que o APHID-V3 apresentou a pior taxa de coincidência com o Alfa-Beta serial. Tal fato demonstra que não basta ter um algoritmo ágil ao realizar um elevado número de podas se a qualidade da solução é ruim. O reflexo disso ocorreria, por exemplo, em um ambiente de disputa, em que o agente indicaria ações ruins e teria um desempenho global insatisfatório. Por outro lado, a versão que apresentou o maior percentual de coincidência foi o APHID-V2 que atingiu a taxa de 81,25% nas execuções que utilizaram a partir de 6 processadores escravos. Neste caso, os limites alfa e beta produziram uma janela de busca mais larga, o que permitiu manter a solução ótima dentro do intervalo.

O APHID-V1 apresentou taxas com valores entre os obtidos pelo APHID-V2 e APHID-V3. No entanto, eles ficaram mais próximos do APHID-V2. Com 6 processadores escravos

o APHID-V1 obteve 75% de coincidência com a solução ótima do Alfa-Beta e, com pelo menos 9 processadores escravos, atingiu a taxa de 68,75%. O ADABA-V2 utiliza o mesmo parâmetro de definição de abertura da janela de busca que o APHID-V1. Neste contexto, apresentou taxas de coincidência superiores que o próprio APHID-V1. A sua melhor atuação ocorreu na execução com 6 processadores escravos em que atingiu 81,25% de coincidência. Nas execuções com um número superior de escravos, o valor de coincidência foi igual a 75%. Estes resultados demonstram que as contribuições do ADABA-V2 trouxeram ganhos significativos para a qualidade da solução.

A taxa de coincidência do ADABA-V1 na execução com 1 processador foi igual à das versões APHID-V1 e APHID-V2. No entanto, quando o número de processadores escravos aumentou, a taxa de coincidência foi inferior à destes algoritmos. Convém salientar que as versões APHID-V1, APHID-V2 e ADABA-V2 utilizaram a definição empírica e manual para formação dos limites da janela de busca. Por outro lado, o ADABA-V1 utilizou uma política de definição automática destes limites, sendo assim, os resultados do ADABA-V1 demonstram que esta política proporciona um cálculo de janela próximo ao do APHID-V1, visto que a diferença entre as taxas de coincidência dessas versões foi igual a 0 para 1 processador escravo, 12,5% para 3 escravos, 31,25% para 6 escravos e 25% nas demais execuções.

O ADABA-V3 apresentou uma taxa de coincidência superior à obtida pelo ADABA-V1 e um pouco inferior ao ADABA-V2. Ressalta-se que o bom desempenho das versões distribuídas deve ser observado pelo balanceamento entre o tempo de execução dos algoritmos e a qualidade da solução retornada. Neste aspecto, se for considerada a eficiência do ADABA-V3 que foi igual a 0,20 com 15 processadores (seção 8.2.4), bem como a taxa de correspondência que foi superior a 60% na maioria das execuções, é possível constatar que a versão proposta neste trabalho apresenta um bom balanceamento.

Em uma visão global da Tabela 12, observa-se que as taxas de coincidência das versões dos algoritmos foram mais elevadas nas execuções que envolveram mais processadores escravos. Isto demonstra que a maior divisão de tarefas, em conjunto com as políticas de prioridade para execução destas tarefas nos processadores escravos, provêm soluções mais relevantes para atualização da borda do processador mestre. Neste sentido, observa-se que não é apenas em relação ao tempo de execução dos algoritmos que o maior número de processadores escravos beneficia, mas também a qualidade da solução.

8.3 Avaliação do monoagente ADABA-Draughts

Esta seção avalia o monoagente jogador de Damas concebido a partir do algoritmo ADABA: o ADABA-Draughts apresentado no Capítulo 6. Para isso, foram produzidas três versões deste jogador correspondentes às versões do ADABA tratadas na seção 8.2. O objetivo foi avaliar a evolução das contribuições inseridas no desenvolvimento do ADABA.

Adicionalmente, esta seção compara as versões do ADABA-Draughts com o agente jogador APHID-Draughts, que utiliza a versão do APHID-V1 - visto que nos testes reportados na seção 8.1 concluiu-se que esta é a versão mais adequada ao problema do jogo de Damas tratado neste trabalho. Desta forma, o desempenho dos agentes jogadores será avaliado em termos do tempo de treinamento dos agentes (seção 8.3.1) e torneios (seção 8.3.2).

8.3.1 Tempo de Treinamento dos Agentes

Esta seção avalia o tempo de treinamento de cada MLP dos agentes automáticos jogadores de Damas ADABA-Draughts e APHID-Draughts. Como no caso particular do ADABA-Draughts serão consideradas três versões, a partir de agora os agentes serão tratados apenas pelo nome do algoritmo (e sua versão) a fim de facilitar a identificação.

Três processos de treinamento foram executados para cada agente variando apenas o limite da profundidade de busca utilizada, isto é, 10, 12 e 14. Um processo foi formado por 100 jogos, sendo 10 seções de 10 jogos seguindo a estratégia de *self-play* com clonagem. Nesta estratégia, conforme apresentado na seção 2.8, ao final de cada seção pode ser gerado um clone da MLP (que obteve o melhor desempenho naquela seção). Ao final de todas as seções, um torneio é disputado entre os clones gerados para, então, indicar a melhor MLP. Como o número de clones gerados pode variar para cada uma das execuções, o tempo de treinamento reportado aqui foi restrito ao tempo consumido por cada agente para executar as 10 seções. Outro fato que pode impactar no tempo de treinamento são os pesos iniciais da rede neural, visto que eles vão comprometer na qualidade da solução retornada pelo algoritmo de busca. Sendo assim, os pesos iniciais de todas as MLPs foram configurados com os mesmos valores. O gráfico da Figura 51 apresenta o tempo obtido em minutos para cada execução.

No geral, observa-se que quanto maior a profundidade de busca utilizada, maior o tempo gasto no processo de treinamento das MLPs. Neste contexto, é importante destacar que quanto maior for a profundidade de busca, mais tempo é dispendido na escolha da solução. Este fato ocorre porque no domínio do jogo de Damas, quanto maior a profundidade de busca, o fator de ramificação da árvore de jogo atinge valores extremos em virtude da presente de peças do tipo Damas, o que eleva a complexidade em relação ao espaço de estados da árvore de busca.

O ADABA-V1 apresentou o maior tempo de treinamento em todas as execuções. Na sequência está o APHID. Observa-se que o tempo consumido por estes dois algoritmos foi muito próximo. Na profundidade 10, a diferença entre o tempo de execução do APHID (365 minutos) e o ADABA-V1 (402 minutos) foi igual a 9%. Na execução cujo limite da profundidade de busca foi igual a 12, esta diferença ficou em 10% (474 minutos para o APHID e 532 minutos para o ADABA-V1). Na profundidade 14 esta diferença chegou a ser inferior a 1%, isto é, 0.45% (1743 minutos para o APHID e 1751 par ao ADABA-V1). Estes resultados reafirmam que a automatização do processo de definição da janela de

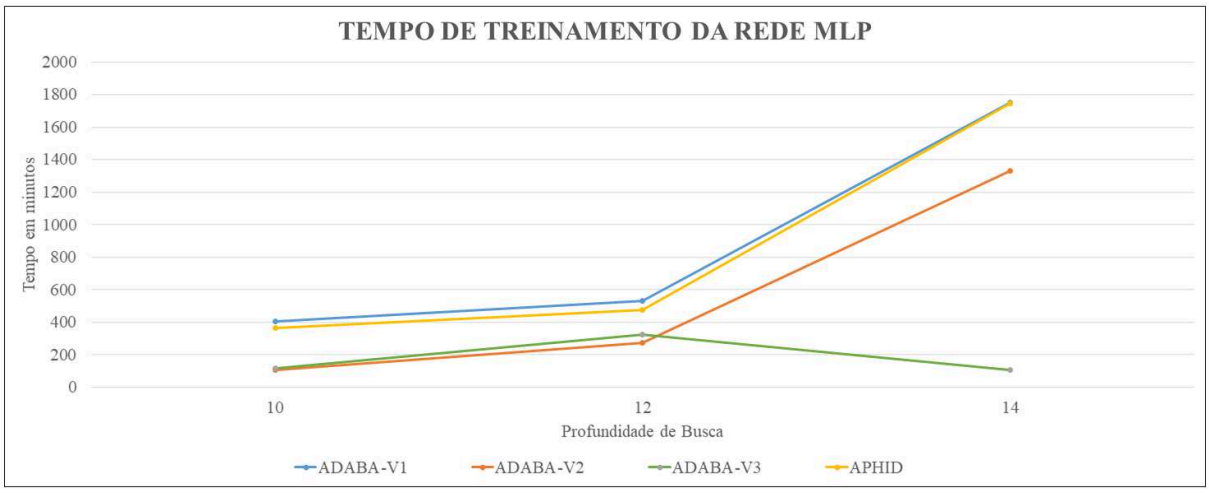


Figura 51 – Tempo de treinamento das redes MLP dos agentes ADABA-Draughts com as versões ADABA-V1, ADABA-V2 e ADABA-V3 e do agente APHID-Draughts com o APHID

busca do ADABA apresentou um tempo de execução bem próximo ao do APHID, tal como descrito na seção 8.2.2.

As versões ADABA-V2 e ADABA-V3 foram mais rápidas que o APHID e ADABA-V1 em todos os casos. O ADABA-V2, com a profundidade de busca limitada a 10, consumiu 29.32% (107 minutos) em relação ao tempo requerido pelo APHID. Neste sentido, o ADABA-V2 foi 70.69% mais rápido. Em relação às profundidades 12 e 14, o ADABA-V2 foi, respectivamente, 43.0% e 23.58% mais rápido do que o agente baseado no APHID, visto que na profundidade de busca igual a 12 foi consumido 270 minutos e, na profundidade 14, 1332 minutos.

O ADABA-V3 foi um pouco mais lento que o ADABA-V2 nas execuções com profundidade de busca igual a 10 e 12. Mais especificamente, na profundidade 10 ele foi 5% mais lento com 113 minutos e, na profundidade 12, ele foi 16% mais lento com 323 minutos. No entanto, na profundidade de busca 14 o tempo de treinamento do ADABA-V3 é superior a todas as versões: ele gastou apenas 107 minutos, chegando a ser mais rápido até do que sua própria execução na profundidade 10. De fato, o ADABA-V3 foi 92% mais rápido do que o ADABA-V2, 93% do que o APHID-V1 e 94% do que o ADABA-V1 na mesma situação (profundidade 14). Sem dúvidas, este é um resultado bastante curioso, mas pode ser justificado pelo fato de que o tempo de treinamento da MLP também está relacionado ao tempo de jogo do conjunto de seções envolvidas no processo de treino. Logo, quanto mais demorado é um jogo, mais lento é o processo de treinamento como um todo. A lentidão de um jogo não está simplesmente vinculada com a profundidade de busca, mas também com o número de jogadas que são realizadas durante uma par-

tida. Existem partidas que entram em um processo de repetição de jogadas, conhecidas como *loop* de final de jogo, que comprometem a agilidade do processo. Neste sentido, foi observado durante o treinamento destas MLPs que os jogos executados pelo ADABA-V3 finalizaram com bem menos jogadas do que os jogos dos demais agentes que, na maioria dos jogos, finalizaram em empate com um número maior de jogadas - mais precisamente, com o limite de jogadas estabelecido pelo sistema para finalizar a partida como empate.

8.3.2 Taxas de vitória obtidas em Torneios

Esta seção avalia a performance dos agentes ADABA-Draughts e APHID-Draughts em termos de taxas de vitória. Para isso, foram realizados três torneios:

1. ADABA-Draughts (atuando com o ADABA-V1) *versus* APHID-Draughts;
2. ADABA-Draughts (atuando com o ADABA-V2) *versus* APHID-Draughts;
3. ADABA-Draughts (atuando com o ADABA-V2) *versus* APHID-Draughts.

Em cada torneio foram executados 40 jogos agrupados em duas rodadas de 20 jogos, que partiram de um conjunto de 20 estados de tabuleiro já iniciados com o mesmo número de peças para cada jogador - trata-se do mesmo conjunto utilizado nos experimentos da seção 8.1.2. Na primeira rodada (Rodada I), o ADABA-Draughts jogou com peças pretas e o APHID-Draughts com peças vermelhas. Na segunda rodada (Rodada II), a situação foi invertida, isto é, o ADABA-Draughts jogou com peças vermelhas e o APHID-Draughts com peças pretas. Neste caso, o objetivo foi não beneficiar nenhum dos agentes jogadores, uma vez que no geral, em jogos como Damas, o agente que inicia a partida (jogador preto) possui mais vantagem estratégica.

Para expansão da árvore de jogo foi utilizada a busca por ID com o tempo limitado a 20 segundos por movimento. Os agentes utilizaram sua rede neural treinada na profundidade 14 no contexto dos experimentos da seção 8.3.1. A seguir, os resultados de cada torneio são comentados.

Torneio 1

Os resultados do Torneio 1 são apresentados na Tabela 13 em que o ADABA-Draughts, operando com o ADABA-V1, enfrentou o APHID-Draughts.

Conforme é possível notar, com peças pretas (Rodada I), o ADABA-Draughts obteve 75% de vitória e 10% de derrotas. Isto corresponde a uma superioridade de 65% em relação ao seu oponente. Atuando com peças vermelhas (Rodada II), o ADABA-Draughts obteve 65% de vitória e 5% de derrota, apresentando uma superioridade de 60% em relação ao seu oponente. Nesta direção, observa-se que o ADABA-V1 teve melhor desempenho jogando tanto com peças pretas quanto vermelhas com uma superioridade muito significativa.

Tabela 13 – Torneio 1: Taxas de vitória obtidas pelo ADABA-Draughts atuando com o ADABA-V1 *versus* o APHID-Draughts

Rodada	**Vitórias**	**Empates**	**Derrotas**
I	75%	15%	10%
II	65%	30%	5%

Tal fato demonstra que a contribuição relacionada a esta versão do ADABA permitiu a criação de uma MLP eficaz e boas tomadas de decisão durante as partidas. Neste aspecto, a Tabela 14 apresenta a taxa de coincidência entre os movimentos escolhidos por cada agente durante as partidas do torneio com os movimentos que o agente Cake realizaria na mesma situação. Ressalta-se que, assim como abordado na seção 8.1.3 o Cake é utilizado aqui como um "conselheiro" de movimentos [78]. Neste aspecto, coerentemente com os resultados reportados na Tabela 13, o ADABA-Draughts apresentou maior coincidência com o Cake em ambas as situações: 55.95% atuando com peças pretas e 59.75% com peças vermelhas.

Tabela 14 – Taxas de coincidência de movimentos com o Cake obtidos nas partidas do Torneio 1

Posição do agente no jogo	ADABA-Draughts	APHID-Draughts
Peças Pretas	55.95%	22.70%
Peças Vermelhas	59.72%	38.74%

Torneio 2

No Torneio 2 o ADABA-Draughts, operando com o ADABA-V2, enfrentou o APHID-Draughts. Os resultados são apresentados na Tabela 15.

Tabela 15 – Torneio 2: Taxas de vitória obtidas pelo ADABA-Draughts atuando com o ADABA-V2 *versus* o APHID-Draughts

Rodada	**Vitórias**	**Empates**	**Derrotas**
I	40%	50%	10%
II	20%	50%	30%

Conforme observado, na Rodada I, o ADABA-Draughts, atuando como jogador preto, obteve as taxas de 40% de vitória e 10% de derrota. Na Rodada II, como jogador vermelho, o ADABA-Dragths obteve a taxa de 20% de vitória e 30% de taxa de derrota. Um ponto a se destacar é que, ambos os jogadores, ao atuarem na posição vantajosa do jogador preto,

foram superiores ao seu oponente (ADABA-Draughts apresentou uma superioridade de 30% e o APHID-Draughts 10%). Nesta direção, os resultados da Tabela 15, também permitem notar que o ADABA-Draughts apresentou uma maior superioridade que o seu oponente quando ambos atuaram com peças pretas.

A Tabela 16 apresenta as taxas de coincidência de movimentos com o jogador Cake obtidas no Torneio 2. Operando com peças pretas, o ADABA-Draughts apresentou uma taxa de coincidência de 62.27% e o APHID-Draughts 45.50%. Por outro lado, operando com peças vermelhas, o ADABA-Draughts apresentou a taxa de 42.50% e o APHID-Draughts 38.50%. Os valores destas taxas além de reforçarem os resultados observados na Tabela 15, permitem notar que o nível de jogo dos agentes ADABA-Draughts e APHID-Draughts neste torneio não foi discrepante. Tal situação refere-se ao fato de que o ADABA-V2 utiliza a mesma política de formação de janela do APHID.

Tabela 16 – Taxas de coincidência de movimentos com o Cake obtidos nas partidas do Torneio 2

Posição do agente no jogo	ADABA-Draughts	APHID-Draughts
Peças Pretas	62,27%	45.50%
Peças Vermelhas	42,50%	38.50%

Torneio 3

A Tabela 17 apresenta os resultados do Torneio 3 em que o ADABA-Draughts operou com o ADABA-V3 e enfrentou o APHID-Draughts. Nela observa-se que atuando com peças pretas, o ADABA-Draughts apresentou as taxas de 90% de vitória e 10% de derrota. Ao atuar com peças vermelhas apresentou uma taxa de vitória igual a 100%.

Tabela 17 – Torneio 3: Taxas de vitória obtidas pelo ADABA-Draughts atuando com o ADABA-V3 *versus* o APHID-Draughts

Rodada	**Vitórias**	**Empates**	**Derrotas**
I	90%	0%	10%
II	100%	0%	0%

Tabela 18 – Taxas de coincidência de movimentos com o Cake obtidos nas partidas do Torneio 3

Posição do agente no jogo	ADABA-Draughts	APHID-Draughts
Peças Pretas	48,39%	26.21%
Peças Vermelhas	67.46%	33.42%

A Tabela 18 apresenta as taxas de coincidência dos jogadores comparados ao Cake. Conforme é possível notar, o ADABA-Draughts apresentou maior taxa de correspondência jogando tanto como preto (48,39% contra 26,21% do APHID-Draughts) quanto vermelho (67,46% contra 33,42% do APHID-Draughts). No caso particular do ADABA-Draughts jogando com peças vermelhas a taxa de coincidência foi superior a 65%, o que corrobora com seu excelente desempenho ao derrotar o APHID-Draughts em todas as partidas da Rodada II, conforme apresentado na Tabela 17.

8.4 Avaliação do multiagente D-MA-Draughts

Esta seção realiza a avaliação do sistema multiagente D-MA-Draughts. Conforme antecipado no Capítulo 7, ele possui uma primeira versão [3] que utiliza o algoritmo YBWC [33] como mecanismo de tomada de decisão. Além disso, esta primeira versão possui duas dinâmica de atuação em partidas. Nesta direção, a seção 8.4.1 apresenta o processo realizado para definir a dinâmica mais adequada para a nova versão do D-MA-Draughts. Na sequência, a seção 8.4.2 apresenta a avaliação do desempenho desta nova versão, que utiliza a versão final do ADABA (V3) como mecanismo de tomada de decisão, através de um torneio disputado contra a sua versão monoagente, o ADABA-Draughts.

8.4.1 Definindo a melhor dinâmica em partidas

Esta seção verifica qual dinâmica de jogo é a mais adequada a representar o D-MA-Draughts em partidas. Como esta avaliação ocorreu logo no início deste trabalho de doutorado, foi utilizada a sua primeira versão apresentada na seção 3.2.2.5. Desta forma, foi realizado um torneio envolvendo as dinâmicas de jogo DI e DII do D-MA-Draughts. O mesmo conjunto de agentes foi empregado na representação das dinâmicas DI e DII. No entanto, após ser atingida a fase de final de jogo, cada jogador irá conduzir a partida segundo a dinâmica que está representando.

O torneio foi composto por 40 jogos. Os jogos foram executados utilizando a estratégia de busca de profundidade limitada com o limite igual a 8. Cada partida do torneio foi realizada a partir de um estado de tabuleiro já iniciado. Tais estados foram obtidos a partir da base de dados de estados de final de jogo utilizado no processo de clusterização do D-MA-Draughts. A fim de não beneficiar nenhum jogador (peças pretas ou vermelhas) foi adotado as seguintes medidas:

1. cada estado de tabuleiro apresentou o mesmo número de peças para cada jogador;
2. cada jogo foi executado duas vezes, sendo que na primeira execução a DI representou o jogador preto e a DII o jogador vermelho e, na segunda execução, esta representação foi invertida, isto é, a DI atuou com o jogador vermelho e a DII com o preto.

Tabela 19 – Torneios executados entre as dinâmicas DI e DII do D-MA-Draughts.

Agente	Vitórias		Empates
	DI	DII	
D-MA-Draughts	5%	22,5%	72,%

A Tabela 19 apresenta os resultados obtidos. Conforme é possível observar, nos jogos em que houve um vencedor, a DII mostrou-se superior com 22.5% de vitórias, enquanto a DI obteve apenas uma taxa de 5% de vitórias. O sucesso da DII prova que o conhecimento especializado dos agentes ao lidar com diferentes perfis de tabuleiro é uma alternativa excelente para jogos como Damas. Além disso, é importante destacar que o ganho em termos de taxas de vitória para o agente que jogou com a DII foi observado mesmo em situações em que o estado de tabuleiro no início do jogo é favorável ao seu oponente. O resultado obtido nesta análise em conjunto com as análises apresentadas em [3] comprovam que a DII é, sem dúvidas, a dinâmica mais adequada para representar o D-MA-Draughts em partidas. Portanto, esta é a dinâmica que será adotada na nova versão do D-MA-Draughts concebida no âmbito deste trabalho.

8.4.2 Atuação do D-MA-Draughts em Disputas

Após a definição da melhor dinâmica de atuação em partidas e da obtenção da versão final do ADABA, foi criada a nova versão do sistema multiagente D-MA-Draughts. Neste caso, tal sistema passa a operar com o algoritmo de alto desempenho ADABA proposto neste trabalho. O IIGA da nova versão do D-MA-Draughts corresponde ao ADABA-Draughts (com o ADABA-V3 dos experimentos da seção 8.3). Os agentes de final de jogo foram treinados segundo o processo descrito na seção 7.4.

Com a intenção de verificar o desempenho do sistema multiagente D-MA-Draughts atuando em um problema prático, um torneio foi realizado entre ele e sua versão monoagente, ADABA-Draughts. A configuração deste torneio foi a mesma aplicada nos experimentos da seção 8.3.2. Desta forma, foram executados 40 jogos agrupados em duas rodadas, sendo que na primeira (Rodada I) o D-MA-Draughts atou com peças pretas e o ADABA-Draughts com peças vermelhas. Na Rodada II, a situação foi invertida. A estratégia de busca foi ID com jogadas limitadas a 20 segundos. A Tabela 20 apresenta as taxas de vitória, derrota e empate obtidas pelo D-MA-Draughts. Além disso, a Tabela 21 apresenta as taxas de correspondência com o "conselheiro" Cake.

Na Tabela 20 observa-se que jogando com peças pretas (Rodada I) o D-MA-Draughts apresentou 70% de taxa de vitória e 5% de taxa de derrota. Na Rodada II, ele obteve 10% de vitória e 70% de derrota. Neste sentido, observa-se que tanto o D-MA-Draughts quanto o ADABA-Draughts tiveram desempenho similar jogando com peças pretas. Na atuação com peças vermelhas, o D-MA-Draughts se saiu um pouco melhor que o ADABA-Draughts na situação equivalente. O reflexo deste resultado é observado na Tabela 21 que

Tabela 20 – Taxas de vitória obtidas pela nova versão do D-MA-Draughts *versus* o monoagente ADABA-Draughts

Rodada	**Vitórias**	**Empates**	**Derrotas**
I	70%	25%	5%
II	10%	20%	70%

Tabela 21 – Taxas de coincidência de movimentos com o Cake obtidos nas partidas do torneio apresentado na Tabela 20

Posição do agente no jogo	D-MA-Draughts	ADABA-Draughts
Peças Pretas	53,25%	44,64%
Peças Vermelhas	36,95%	39,03%

mostra que, jogando com peças pretas, o D-MA-Draughts apresentou uma taxa de correspondência igual a 53,25% com o Cake e o ADABA-Draughts 44,64%. Com peças vermelhas o D-MA-Draughts teve 36.95% e o ADABA-Draughts 39.03%. De fato, o resultado deste torneio foi muito equilibrado para ambos os jogadores. O motivo disso é que o D-MA-Draughts parte da mesma rede neural que o ADABA-Draughts. Neste sentido, até que a partida atinja o nível de final de jogo, é como se os agentes estivessem jogando contra si próprio. A partir do estado de final de jogo, configurado com 12 peças ou menos, o D-MA-Draughts inicia sua atuação com a dinâmica DII (seção 7.3.2) pressionando o seu adversário. Por esta razão, houve um número significativo de empates: 25% na Rodada I e 20% na Rodada II.

8.5 Considerações Finais

Este Capítulo apresentou todos os experimentos realizados no âmbito deste trabalho de doutorado. Para isso, foram estabelecidas quatro etapas: 1) comparação do desempenho das abordagens síncrona e assíncrona - representadas pelas versões distribuídas do algoritmo Alfa-Beta YBWC (síncrona) e APHID (assíncrona) - em um problema prático; 2) avaliação do desempenho do algoritmo proposto neste trabalho, o ADABA; 3) avaliação do monoagente baseado no algoritmo proposto, o ADABA-Draughts; 4) avaliação SMA D-MA-Draughts.

Na primeira etapa, foram realizados experimentos com o agente D-VisionDraughts que atua em ambiente de alto desempenho utilizando a versão com abordagem síncrona de paralelismo YBWC, e o agente APHID-Draughts, desenvolvido no contexto deste trabalho. A implementação e os experimentos foram realizados em arquitetura de memória distribuída, que, sem dúvidas, representa um desafio para a distribuição do Alfa-Beta. De fato, a principal intenção foi comparar o desempenho das abordagens síncrona e as-

síncrona de paralelismo aplicadas a um mesmo problema prático. Nesta direção, foram considerados os seguintes parâmetros: tempo de treinamento, taxas de vitória e taxa de coincidência entre a escolha de movimentos executados por cada agente (APHID-Draughts e D-VisionDraughts) e aqueles que o "conselheiro" Cake realizaria nas mesmas situações. Os resultados obtidos mostraram que a abordagem assíncrona é mais adequada que a versão síncrona para agentes de Damas quando implementadas em arquitetura de memória distribuída.

Na segunda etapa, foi avaliada a proposta da nova versão distribuída do Alfa-Beta, o ADABA. Para isso, o ADABA foi comparado com o APHID. Para cada algoritmo foi estabelecida três versões (V1, V2 e V3). No caso do ADABA, o objetivo destas versões foi avaliar a evolução das contribuições inseridas no algoritmo proposto. No APHID, a intenção foi verificar o comportamento do algoritmo ao variar o parâmetro de abertura da janela de busca que é determinado de modo empírico e manual. Os resultados mostraram que, de fato, a política do APHID em determinar o parâmetro de abertura da janela de busca de modo empírico e manual representa uma fragilidade. Neste contexto, se este valor é grande (como ocorreu com o APHID-V2), corre-se o risco de impactar na velocidade do algoritmo agravando a sobrecarga de busca. Por outro lado, se o valor é muito pequeno (como o APHID-V3), perde-se com frequência a solução ótima, o que prejudica o desempenho global do algoritmo de busca em relação à qualidade da solução. O ADABA apresentou bons resultados em todas as suas versões, sendo que a sua versão final (ADABA-V3) apresentou um bom equilíbrio entre a sobrecarga de busca, tempo de execução e qualidade da solução da busca. Além disso, a política automática de formação do parâmetro de abertura da janela de busca proposta pelo ADABA eliminou a fragilidade identificada no APHID. Nos experimentos também foi possível observar que o bom desempenho de uma versão distribuída do Alfa-Beta não pode ser atribuída apenas à aceleração do algoritmo, sendo necessário haver um equilíbrio com a qualidade da solução.

Na terceira etapa, o desempenho dos algoritmos ADABA e APHID foram avaliados em uma aplicação prática, isto é, na obtenção de agentes jogadores de Damas para disputarem torneios. Assim, houve o treinamento das redes MLP dos agentes envolvidos nos experimentos. Na sequência, os agentes treinados baseados no ADABA (ADABA-Draughts) disputaram torneios contra o APHID-Draughts. Os resultados mostraram que a versão final do ADABA (V3) apresentou bom desempenho em relação ao tempo de treinamento do agente jogador e uma taxa de vitórias bem elevada nas partidas que disputou contra o APHID. Tal fato demonstra que as contribuições inseridas no ADABA em conjunto produziram uma MLP de qualidade e permitiram que o agente realizasse, na prática, tomadas de decisões mais acuradas (este fato pode ser notado pela elevada taxa de coincidência do jogador baseado no ADABA-V3 com o renomado jogador Cake).

A última etapa dos experimentos avaliou o desempenho do SMA D-MA-Draughts. Primeiramente, foi definida a melhor dinâmica de atuação em partidas. Os resultados

mostraram que a dinâmica em que houve maior cooperação entre os agentes, a DII, apresentou melhor desempenho. Tal situação foi observada mesmo em situações onde, para um mesmo estado de tabuleiro, a DII atuou representando as peças pretas ou vermelhas. Este fato é justificado, pois após alguns movimentos outro agente pode representar melhor o perfil do tabuleiro e tomar a ação mais adequada. Portanto, a dinâmica DII foi a adotada na nova versão do D-MA-Draughts. Na sequência, foi avaliado o desempenho do D-MA-Draughts em um torneio contra sua versão monoagente. Os resultados mostraram um equilíbrio entre estes jogadores, visto que, até atingir o estado de final de jogo, ambos agentes atuam com a mesma rede MLP.

Capítulo 9

Conclusão

Este trabalho apresentou a proposta de uma nova versão de distribuição do algoritmo de busca Alfa-Beta, o ADABA. A fim de validar o desempenho desta proposta, o domínio do jogo de Damas foi utilizado por meio da produção do SMA D-MA-Draughts.

Neste contexto, o processo de implementação do ADABA envolveu uma fase inicial que foi caracterizada pelo estudo teórico e prático entre as abordagens síncrona e assíncrona de distribuição do algoritmo de busca Alfa-Beta. Para isso, foram utilizados os algoritmos YBWC (síncrono) e APHID (assícrono). A avaliação prática analisou o comportamento destes algoritmos aplicados a um mesmo problema de elevada complexidade, ou seja, o domínio do jogo de Damas. Assim, foi utilizado um sistema monoagente automático jogador de Damas já implementado pela autora deste trabalho em arquitetura de memória distribuída, o chamado D-VisionDraughts, que emprega o algoritmo YBWC como mecanismo de tomada de decisão. Além disso, para realizar a comparação com o APHID, foi necessário implementar tal algoritmo no contexto deste trabalho e inseri-lo em uma arquitetura análoga à do D-VisionDraughts - o agente produzido foi denominado APHID-Draughts. Desta forma, foram relacionados pontos fortes e limitações de cada algoritmo a fim de nortear a proposta do ADABA que seria produzida posteriormente. Nos experimentos da avaliação prática, foi possível observar que o APHID se mostrou mais adequado a atuar em arquitetura de memória distribuída para o caso do problema do jogo de Damas.

O ADABA implementa o modelo mestre-escravo de paralelismo baseado na abordagem assíncrona de distribuição do APHID. De fato, o ADABA teve por objetivo melhorar o desempenho do APHID por meio das seguintes contribuições: 1) implementar uma política automatizada para formação da janela de busca dos processadores escravos; 2) estabelecer uma nova política de priorização das tarefas que serão executadas nos processadores escravos baseada na proximidade da solução desejada; e 3) aumentar o número de *threads* do mestre dedicadas a tratar as tarefas recebidas dos processadores escravos. No caso particular da contribuição referente à proposta da política da janela de busca, o ADABA empregou conceitos inerentes à abordagem síncrona do YBWC, que cria uma

janela de busca inicial baseada no irmão mais a esquerda da árvore de busca.

O desempenho do ADABA foi avaliado por meio da comparação com o algoritmo APHID. Para isso, foi construído o monoagente jogador de Damas ADABA-Draughts - correspondente a um dos agentes do SMA D-MA-Draughts - e utilizado o jogador APHID-Draughts. Ambos jogadores atuaram em arquitetura de memória distribuída. Os algoritmos ADABA e APHID foram testados em três situações distintas, sendo que no caso no ADABA a intenção foi avaliar o desempenho do algoritmo por meio do agrupamento das contribuições, sendo que foi criada uma versão com a primeira contribuição (ADABA-V1), uma outra versão (ADABA-V2) com as demais contribuições (2 e 3), e a terceira versão (ADABA-V3) como a junção das versões V1 e V2.

No caso do APHID, também foram criadas três versões, que se distinguiram apenas em relação ao parâmetro, definido empírica e manualmente, utilizado para estabelecer os limites da janela de busca. Sendo assim, a intenção foi analisar o comportamento do algoritmo na utilização de um valor para esse parâmetro considerado bom (APHID-V1), de um valor grande (APHID-V2) e de um valor pequeno (APHID-V3). Os resultados dos experimentos mostraram que, de fato, a política de definição de janela do APHID constitui uma fragilidade do algoritmo, visto que, a janela pode ficar muito larga, gerando sobrecarga de busca elevada (como pode ser observado na versão APHID-V2), ou muito estreita, prejudicando a qualidade da solução do algoritmo (conforme observado na versão APHID-V3). Em relação ao ADABA, foi verificado que a sua nova política de definição da janela de busca (empregada na versão ADABA-V1) conseguiu eliminar esta fragilidade encontrada no APHID. A versão ADABA-V2 atingiu uma melhor velocidade da busca quando utilizado um número superior a um processador escravo. A versão ADABA-V3 mostrou que as contribuições do algoritmo proposto, em conjunto, permitiram um bom equilíbrio entre as fontes de ineficiência (sobrecargas de busca e comunicação) e acurácia da solução retornada.

A arquitetura final do ADABA-Draughts, formada pela junção de suas contribuições, foi então expandida para todos os agentes do D-MA-Draughts. É importante ressaltar que o D-MA-Draughts possui uma primeira versão que conta com duas dinâmicas de atuação em partidas, sendo que a primeira dinâmica possui menor integração entre os agentes e, a segunda, maior integração entre eles. Neste sentido, ainda no início deste trabalho de doutorado, foi realizado um experimento a fim de definir a melhor dinâmica para este SMA. Desta forma, foi concluído que a dinâmica em que há maior cooperação entre os agentes é a mais adequada para produzir bons resultados em partidas e passou a ser adotada na nova versão do D-MA-Draugths. Foram conduzidos experimentos colocando em perspectiva o D-MA-Draughts *versus* sua versão monoagente ADABA-Draughts. Os resultados mostraram um equilíbrio nas taxas de vitória obtidas. Neste sentido, foi possível concluir que a arquitetura do D-MA-Draugths foi fortalecida pela inclusão do ADABA, visto que, nos testes conduzidos no âmbito deste trabalho, tal algoritmo apresentou desempenho

satisfatório.

9.1 Contribuições Científicas

As principais contribuições do presente trabalho são:

- ❏ Criação de uma nova versão distribuída assíncrona do algoritmo Alpha-Beta, denominada ADABA, que é inspirada no APHID e que conta com estratégias para reduzir as fragilidades do algoritmo APHID;
- ❏ Criação de uma biblioteca de busca composta pelo algoritmo Alfa-Beta versão *Fail-Soft* e pela versão distribuída proposta neste trabalho, o ADABA. Tal biblioteca permite o emprego destes algoritmos em outros sistemas que utilizem o Alfa-Beta para resolução de problemas. Desta forma, existe a opção de escolha entre a versão serial e distribuída.
- ❏ Criação do sistema monoagente jogador de Damas automático APHID-Draughts cujo mecanismo de busca conta com a abordagem assíncrona de paralelismo do Alfa-Beta (APHID).
- ❏ Criação do sistema monoagente jogador de Damas automático ADABA-Draughts cujo mecanismo de busca conta com a abordagem assíncrona de paralelismo do Alfa-Beta proposta neste trabalho (ADABA).
- ❏ Obtenção de uma plataforma multiagente, o D-MA-Draughts, que estende a arquitetura do ADABA-Draughts a todos os agentes e utiliza a melhor dinâmica de cooperação entre os agentes em partidas.

É importante destacar que o algoritmo produzido no contexto deste trabalho, o ADABA, pode ser estendido a outros problemas que sejam caracterizados pela perspectiva de que as ações de um agente venham a minimizar os efeitos das ações de um outro agente, assim como ocorre nos jogos do tipo Soma Zero. Para isso, é necessário a implementação dos métodos de geração dos estados sucessores da árvore de busca, bem como da função de avaliação. Tais requisitos são necessários, pois são casos particulares do problema tratado.

9.2 Limitações

Algumas das limitações deste trabalho são descritas a seguir:

1. O ADABA foi projetado de modo a operar tanto em arquitetura de memória distribuída, quanto compartilhada. Todavia, como não havia infra-estrutura de memória

compartilhada, não foi possível realizar experimentos do algoritmo nesta arquitetura, fato que impossibilitou, inclusive, a comparação com outros resultados reportados na literatura, visto que todos os trabalhos correlatos que propuseram versões distribuídas do Alfa-Beta as avaliaram em arquitetura de memória compartilhada.

2. A arquitetura de memória distribuída utilizada foi limitada a um número pequenos de máquinas, não permitindo o avaliação do algoritmo por "stress" a fim de verificar outras combinações de distribuição para a busca que permitissem concluir a quantidade adequada de processos escravos por máquina, bem como, o limite de processos escravos que não degradariam o desempenho global do algoritmo devido ao agravamento de algumas fontes de ineficiência como a sobrecarga de comunicação.

9.3 Produção Bibliográfica

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

1. Artigo completo em conferência referente ao primeiro objetivo específico da seção 1.3.1: "TOMAZ, L. B. P; JULIA, R. M. S.; FARIA, Matheus P. P. *APHID-Draughts: comparing the synchronous and asynchronous parallelism approaches for the Alpha-Beta algorithm applied to Checkers.* In: 2017 IEEE 29th Conference on Tools with Artificial Intelligence (ICTAI), Boston, MA, USA. DOI: 10.1109/ICTAI.2017.00188" (Ciência da Computação: Qualis B1);

2. Artigo completo aceito em conferência referente ao segundo objetivo específico da seção 1.3.1: "TOMAZ, L. B. P; JULIA, R. M. S. *ADABA: an Algorithm to Improve the Parallel Search in Competitive Agents.* In: 18th Internacional Conference on Intelligent Systems Design and Applications, Vellore, India, 2018" (Ciência da Computação: Qualis B1) - a conferência ocorrerá entre os dias 6 e 8 de dezembro de 2018;

3. Artigo de revista referente ao quarto objetivo específico da seção 1.3.1: "TOMAZ, L. B. P; JULIA, R. M. S; DUARTE, V. A. R. *A multiagent player system composed by expert agents in specific game states operating in high performance environment.* Journal of Applied Inteliggence (APIN), Springer US, v. 48 n. 1, p. 1-22. DOI: 10.1007/s10489-017-0952-x" (Ciência da Computação: Qualis B1);

O artigo listado abaixo, também refere-se ao quarto objetivo listado na seção 1.3.1, mas está em processo de submissão:

1. Artigo de revista referente ao quarto objetivo específico da seção 1.3.1: "TOMAZ, L. B. P; JULIA, R. M. S. *ADABA: Improving the Balancing Between Synchronization Overhead and Acuracy in a New Distributed Version of the Alpha-Beta Algorithm.*

O artigo listado abaixo foi produzido como resultado do trabalho de Iniciação Científica PIBIC-FAPEMIG do qual a autora deste trabalho atuou como coorientadora.

1. "FARIA, M. P. P; JULIA, R. M. S.; TOMAZ, L. B. P. *Proposal of an automatic tool for evaluating the quality for decision-making on checkeres player agents.* In: Proceedings do Encontro Nacional de Inteligência Artificial e Computacional (ENIAC) 2018, 2018. Brasília, Brasil." (Ciência da Computação: Qualis B4).

9.4 Trabalhos Futuros

Durante o desenvolvimento desta pesquisa, algumas questões foram levantadas para observar o desempenho da algoritmo proposto, bem como melhorar a arquitetura jogadora de Damas utilizada no contexto do trabalho. 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:

- ❏ Expandir a avaliação do algoritmo ADABA empregando parâmetros referentes ao tempo de comunicação (troca de mensagens) entre os processadores e tempo de CPU consumido efetivamente a fim de verificar possíveis desperdícios de recursos que poderiam ocorrer em detrimento da arquitetura distribuída empregada no ADABA.

- ❏ Incluir uma programação multi-processo baseada, por exemplo, na interface OpenMP [83], para criar um cenário de paralelismo local usufruindo da memória compartilhada disponível em cada estação de trabalho da arquitetura de memória distribuída (que é constituída por diversas máquinas).

- ❏ Substituir a interface de troca de mensagens empregada no ADABA - RMI (*Remote Message Interface*) [81] - pela MPI (*Message Passing Inteface*) [84] a fim de verificar se há um melhor desempenho em tempo de execução do algoritmo.

- ❏ Expandir a utilização do ADABA para outros domínios de problemas de alta complexidade como o jogo de Xadrez. Ressalta-se que em Ref. [14] os autores propuseram uma arquitetura que mostrou produzir um agente de alto desempenho em poucas horas. No entanto, a arquitetura de hardware utilizada possui um poder computacional muito expressivo, inacessível para diversos grupos de pesquisa. Neste sentido, observa-se que a utilização da arquitetura distribuída por meio de estações de trabalho convencionais pode contribuir para o avanço de pesquisas que buscam soluções inteligentes para problemas com um espaço de estados elevadíssimo.

- Substituir a arquitetura da MLP adotada pelo agente ADABA-Draughs 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 método atual em que é utilizado um conjunto de características inerentes ao próprio jogo de damas por meio do mapeamento NET-FEATUREMAP.

Referências

1 BROCKINGTON, M. G. *Asynchronous Parallel Game-Tree Search*. Tese (Doutorado), Edmonton, Alta., Canada, 1998. AAINQ29023.

2 SUTTON, R. S.; BARTO, A. G. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. [S.l.]: The MIT Press, 1998. ISBN 0262193981.

3 TOMAZ, L. B. P. *D-MA-Draughts: Um Sistema Multiagente Jogador de Damas Automático que Atua em um Ambiente de Alto Desempenho*. Dissertação (Mestrado) — Universidade Federal de Uberlândia, 2013.

4 CAMPOS, P.; LANGLOIS, T. Abalearn: A risk-sensitive approach to self-play learning in abalone. In: *Proceedings of the 14th European Conference on Machine Learning*. Berlin, Heidelberg: Springer-Verlag, 2003. (ECML'03), p. 35–46. ISBN 3-540-20121-1, 978-3-540-20121-2. Disponível em: <https://doi.org/10.1007/978-3-540-39857-8_6>.

5 MITCHELL, T. M. *Machine Learning*. 1. ed. New York, NY, USA: McGraw-Hill, Inc., 1997. ISBN 0070428077, 9780070428072.

6 NEUMANN, J. V.; MORGENSTERN, O. *Theory of Games and Economic Behavior*. [S.l.: s.n.], 1944.

7 DUTTA, P. K. *Strategies and Games: Theory and Practice*. 1. ed. Cambridge: The MIT Press, 1999. v. 1. Disponível em: <https://EconPapers.repec.org/RePEc:mtp:titles:0262041693>.

8 SILVER, D.; HUANG, A.; MADDISON, C. J.; GUEZ, A.; SIFRE, L.; DRIESSCHE, G. van den; SCHRITTWIESER, J.; ANTONOGLOU, I.; PANNEERSHELVAM, V.; LANCTOT, M.; DIELEMAN, S.; GREWE, D.; NHAM, J.; KALCHBRENNER, N.; SUTSKEVER, I.; LILLICRAP, T.; LEACH, M.; KAVUKCUOGLU, K.; GRAEPEL, T.; HASSABIS, D. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, Nature Publishing Group, v. 529, n. 7587, p. 484–489, jan 2016.

9 HERIK, H. J. van den; UITERWIJK, J. W. H. M.; RIJSWIJCK, J. van. Games Solved: Now and In The Future. *Artif. Intell.*, Elsevier Science Publishers Ltd., v. 134, n. 1-2, p. 277–311, jan 2002. ISSN 0004-3702.

10 RUSSELL, S.; NORVIG, P. *Inteligência Artificial - Uma abordagem Moderna*. 2nd. ed. [S.l.]: Campus, 2004.

11 CAMPBELL, M.; HOANE, A. J.; HSU, F. Deep Blue. *Artificial Intelligence*, Elsevier Science Publishers Ltd., Essex, UK, v. 134, n. 1-2, p. 57–83, jan 2002. ISSN 0004-3702. Disponível em: <https://doi.org/10.1016/S0004-3702(01)00129-1>.

12 URA, A.; YOKOYAMA, D.; CHIKAYAMA, T. Two-Level Task Scheduling for Parallel Game Tree Search Based on Necessity. *Information and Media Technologies*, v. 8, n. 1, p. 32–40, 2013. Disponível em: <https://doi.org/10.2197/ipsjjip.21.17>.

13 URA, A.; TSURUOKA, Y.; CHIKAYAMA, T. Dynamic Prediction of Minimal Trees in Large-Scale Parallel Game Tree Search. *Journal of Information Processing*, v. 23, n. 1, p. 9–19, 2015. Disponível em: <https://doi.org/10.2197/ipsjjip.23.9>.

14 SILVER, D.; HUBERT, T.; SCHRITTWIESER, J.; ANTONOGLOU, I.; LAI, M.; GUEZ, A.; LANCTOT, M.; SIFRE, L.; KUMARAN, D.; GRAEPEL, T.; LILLICRAP, T. P.; SIMONYAN, K.; HASSABIS, D. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR*, abs/1712.01815, 2017.

15 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. Disponível em: <https://doi.org/10.1109/CEC.2000.870729>.

16 SCHAEFFER, J.; BURCH, N.; BJORNSSON, Y.; KISHIMOTO, A.; MULLER, M.; LAKE, R.; LU, P.; SUTPHEN, S. Checkers is Solved. *Science*, 2007. ISSN 1144079+.

17 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. *SBIA 2008, LNAI Springer Verlag, New York*, v. 5249, p. 73–82, 2008.

18 NETO, H. C.; JULIA, R. M. S.; CAEXETA, G. S.; BARCELOS, A. R. A. LS-VisionDraughts: Improving the Performance of an Agent for Checkers by Integrating Computational Intelligence, Reinforcement Learning and a Powerful Search Method. *Applied Intelligence*, v. 41, n. 2, p. 525–550, 2014. Disponível em: <http://dx.doi.org/10.1007/s10489-014-0536-y>.

19 TOMAZ, L. B. P.; JULIA, R. S.; BARCELOS, A. R. A. Improving the Accomplishment of a Neural Network Based Agent for Draughts that Operates in a Distributed Learning Environment. *IEEE 14th International Conference on Information Reuse and Integration*, 2013. Disponível em: <https://doi.org/10.1109/IRI.2013.6642481>.

20 DUARTE, V. A. R.; JULIA, R. M. S.; ALBERTINI, M. K.; NETO, H. C. MP-Draughts: Unsupervised Learning Multi-agent System Based on MLP and Adaptive Neural Networks. In: *27th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2015, Vietri sul Mare, Italy, November 9-11, 2015*. [s.n.], 2015. p. 920–927. Disponível em: <http://dx.doi.org/10.1109/ICTAI.2015.133>.

21 TOMAZ, L. B. P.; JULIA, R. M. S.; DUARTE, V. A. A Multiagent Player System Composed by Expert Agents in Specific Game Stages Operating in High Performance Environment. *Appied Inteligence*, v. 48, n. 1, p. 1–22, Jan 2017. Disponível em: <https://doi.org/10.1007/s10489-017-0952-x>.

22 TOMAZ, L. B. P.; JULIA, R. M. S.; FARIA, M. P. P. APHID-Draughts: Comparing the Synchronous and Asynchronous Parallelism Approaches for the Alpha-Beta Algorithm Applied to Checkers. In: *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. [S.l.: s.n.], 2017. p. 1243–1250. ISSN 2375-0197.

23 BROCKINGTON, M. G. *KEYANO Unplugged – The Construction of an Othello Program*. Tr 97-05. [S.l.], 1997.

24 JUNIOR, E. V.; JULIA, R. M. S. BTT-Go: An Agent for Go that Uses a Transposition Table to Reduce the Simulations and the Supervision in the Monte-Carlo Tree Search. In: *FLAIRS Conference*. [S.l.: s.n.], 2014.

25 SANTOS, G. M.; JULIA, R. M. S. Go-Ahead: Improving Prior Knowledge Heuristics by Using. In: *Proceedings of the 29th Florida Artificial Intelligence Research Society Conference*. [S.l.: s.n.], 2016.

26 SILVER, D.; SCHRITTWIESER, J.; SIMONYAN, K.; ANTONOGLOU, I.; HUANG, A.; GUEZ, A.; HUBERT, T.; BAKER, L.; LAI, M.; BOLTON, A.; CHEN, Y.; LILLICRAP, T.; HUI, F.; SIFRE, L.; DRIESSCHE, G. van den; GRAEPEL, T.; HASSABIS, D. Mastering the Game of Go Without Human Knowledge. v. 550, p. 354–359, 10 2017.

27 KNUTH, D. E.; MOORE, R. W. An Analysis of Alpha-Beta Pruning. In: *Artificial Intelligence*. [S.l.: s.n.], 1975. p. 293–326. ISSN 0004-3702.

28 ATKIN, L.; SLATE, D. Computer Chess Compendium. In: LEVY, D. (Ed.). Berlin, Heidelberg: Springer-Verlag, 1988. cap. Chess 4.5-The Northwestern University Chess Program, p. 80–103. ISBN 0-387-91331-9. Disponível em: <http://dl.acm.org/citation.cfm?id=61701.67010>.

29 DUARTE, V. A. R.; JULIA, R. M. S.; BARCELOS, A. R. A.; OTSUKA, A. B. MP-Draughts: a Multiagent Reinforcement Learning System Based on MLP and Kohonen-SOM Neural Networks. *IEEE International Conference on Systems, Man, and Cybernetics*, 2009. Disponível em: <https://doi.org/10.1109/ICSMC.2009.5345960>.

30 MARSLAND, T. A.; CAMPBELL, M. Parallel Search of Strongly Ordered Game Trees. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 14, n. 4, p. 533–551, dec 1982. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/356893.356895>.

31 HYATT, R. M. *A High Performance Parallel Algorithm to Search Depth-first Game Trees*. Tese (Doutorado), 1988. AAI8909785.

32 NEWBORN, M. Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search. *IEEE Trans. Pattern Anal. Mach. Intell.*, IEEE Computer Society, Washington, DC, USA, v. 10, n. 5, p. 687–694, sep 1988. ISSN 0162-8828. Disponível em: <https://doi.org/10.1109/34.6777>.

33 FELDMAN, R.; MONIEN, B.; MYSLIWIETZ, P.; VORNBERGER, O. Parallel Algorithms for Machine Intelligence and Vision. In: KUMAR, V.; GOPALAKRISHNAN, P. S.; KANAL, L. N. (Ed.). New York, NY, USA: Springer-Verlag New York, Inc., 1990. cap. Distributed Game Tree Search, p. 66–101. ISBN 0-387-97227-7. Disponível em: <http://dl.acm.org/citation.cfm?id=107225.107234>.

34 FELDMANN, R.; MONIEN, B.; MYSLIWIETZ, P.; VORNBERGER, O. Distributed Game Tree Search. *IICA Journal*, v. 12, n. 2, p. 65–73, 1989.

35 HAYKIN, S. *Neural Networks: A Comprehensive Foundation*. 2nd edition. ed. [S.l.]: Printice Hall, 1998.

36 WOOLDRIDGE, M. *An Introduction to Multiagent Systems*. 2. ed. New York, NY, USA: John Wiley & Sons, Inc., 2009.

37 CAO, Y.; YU, W.; REN, W.; CHEN, G. An Overview of Recent Progress in the Study of Distributed Multi-Agent Coordination. *IEEE Transactions on Industrial informatics*, IEEE, v. 9, n. 1, p. 427–438, 2013.

38 ROSACI, D. CILIOS: Connectionist Inductive Learning and Inter-Ontology Similarities for Recommending Information Agents. *Information systems*, Elsevier, v. 32, n. 6, p. 793–825, 2007.

39 ROSACI, D.; SARNÉ, G. M. L. EVA: An Evolutionary Approach to Mutual Monitoring of Learning Information Agents. *Applied Artificial Intelligence*, Taylor & Francis, v. 25, n. 5, p. 341–361, 2011. Disponível em: <https://doi.org/10.1080/08839514.2011.559907>.

40 _____. Cloning Mechanisms to Improve Agent Performances. *Journal of Network and Computer Applications*, Elsevier, v. 36, n. 1, p. 402–408, 2013. Disponível em: <https://doi.org/10.1016/j.jnca.2012.04.018>.

41 GOLPAYEGANI, F.; DUSPARIC, I.; TAYLOR, A.; CLARKE, S. Multi-Agent Collaboration for Conflict Management in Residential Demand Response. *Computer Communications*, Elsevier, v. 96, p. 63–72, 2016. Disponível em: <https://doi.org/10.1016/j.comcom.2016.04.020>.

42 MOULIN, B.; BRAHIM, C.-D. An Overview of Distributed Artificial Intelligence. *Foundations of Distributed Artificial Intelligence*, 1996.

43 MARSLAND, T. A. A Review of Game-Tree Pruning. *ICCA Journal*, v. 1, p. 3–19, 1986.

44 FREY, P. W. *Chess Skill in Man and Machine*. New York :: Springer-Verlag„ 1978.

45 MILLINGTON, I. Artificial Intelligence for Games. *Morgan Kaufmann Publishers Inc.*, 2006.

46 PLAAT, A. *Research Re: search & Re-search*. Tese (Doutorado) — Rotterdam, Netherlands, 1996.

47 SHAMS, R.; KAINDL, H. Using Aspiration Windows for Minimax Algorithms. Kaufmann, p. 192–197, 1991.

48 MANOHARARAJAH, V. *Parallel Alpha-Beta Search on Shared Memory Multiprocessors*. Dissertação (Mestrado) — University of Toronto, 2001.

49 BARCELOS, A. R. A. *D-VisionDraughts: Uma Rede Neural Jogadora de Damas que Aprende por Reforço em um Ambiente de Computação Distribuída*. Dissertação (Mestrado) — Universidade Federal de Uberlândia, 2011.

50 ARNESON, E. *How To Play Checkers - Standard U.S. Rules*. 2017. Disponível em: <https://www.thespruce.com/play-checkers-using-standard-rules-409287>. Acesso em: 07/09/2017.

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

52 SAMUEL, A. L. Some Studies in Machine Learning Using the Game of Checkers. *IBM J. Res. Dev.*, IBM Corp., Riverton, NJ, USA, v. 3, n. 3, p. 210–229, jul 1959. ISSN 0018-8646. Disponível em: <https://doi.org/10.1147/rd.33.0210>.

53 MCCULLOCH, W.; PITTS, W. A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, v. 5, p. 115–133, 1943. Disponível em: <https://doi.org/10.1007/BF02478259>.

54 KOHONEN, T. The Self-Organizing Map. *Proceedings of the IEEE*, v. 78, n. 9, p. 1464 –1480, sep 1990. ISSN 0018-9219.

55 _____. *Self-Organizing Maps*. [S.l.]: Springer, 2001.

56 KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W. Reinforcement Learning: a Survey. *Journal of Artificial Intelligence Research*, v. 4, p. 237–285, 1996. Disponível em: <https://doi.org/10.1613/jair.301>.

57 MARTINS, W.; AFONSECA, U. R.; NALIN, L. E. G.; GOMES, V. M. Tutoriais Inteligentes Baseados em Aprendizado por Reforço: Concepção, Implementação e Avaliação Empírica. *Simpósio Brasileiro de Informática na Educação - SBIE - Mackenzie*, 2007.

58 HARMON, M. E.; HARMON, S. S. *Reinforcement Learning: A Tutorial*. 1996.

59 SUTTON, R. S. Learning to Predict by the Methods of Temporal Differences. *Mach. Learn.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 3, n. 1, p. 9–44, aug 1988. ISSN 0885-6125. Disponível em: <https://doi.org/10.1023/A:1022633531479>.

60 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.

61 STRNAD, D.; GUID, N. Parallel Alpha-Beta Algorithm on the GPU. *Journal of Computing and Information Technology*, v. 4, p. 269–274, 2011. Disponível em: <https://doi.org/10.2498/cit.1002029>.

62 SCHAEFFER, J.; LAKE, R.; LU, P.; BRYANT, M. CHINOOK: The World Man-Machine Checkers Champion. *AI Magazine 17*, p. 21–29, 1996.

63 FIERZ, M. C. *Cake Informations*. 2016. Http://www.fierz.ch/cake.php (Disponível em 22/04/2017).

64 CHECKERBOARD. 2017. Disponível em: <http://www.fierz.ch/checkerboard.php>. Acesso em: 07/09/2017.

65 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>.

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

67 FOGEL, D. B.; CHELLAPILLA, K. Verifying Anaconda's Expert Rating by Competing Against Chinook: Experiments in Co-Evolving a Neural Checkers Player. *Neurocomputing*, v. 42, n. 1-4, p. 69–86, 2002. Disponível em: <https://doi.org/10.1016/S0925-2312(01)00594-X>.

68 CHEHELTANI, S. H.; EBADZADEH, M. M. Immune Based Fuzzy Agent Plays Checkers Game. *Appl. Soft Comput.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 12, n. 8, p. 2227–2236, aug 2012. ISSN 1568-4946. Disponível em: <https://doi.org/10.1016/j.asoc.2012.03.009>.

69 AL-KHATEEB, B.; KENDALL, G. Introducing Individual and Social Learning Into Evolutionary Checkers. *IEEE Transactions on Computational Intelligence and AI in Games*, v. 4, n. 4, p. 258–269, 2012. ISSN 1943-068X.

70 _____. Effect of Look-Ahead Depth in Evolutionary Checker. *Journal of the Computer Science and Technology*, v. 5, n. 27, p. 996–1006, 2012.

71 FOGEL, D. B. *Blondie24: Playing at the Edge of AI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN 1-55860-783-8. Disponível em: <https://doi.org/10.1016/B978-155860783-5/50016-7>.

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

73 BREUKER, D.; UITERWIJK, J.; HERIK, H. Information in Transposition Tables. v. 8, 02 1970.

74 BREUKER, D. M.; UITERWIJK, J. W. H. M.; HERIK, H. J. V. D. Replacement Schemes for Transposition Tables. *ICCA Journal*, v. 17, p. 183–193, 1994. Disponível em: <https://doi.org/10.3233/ICG-1994-17402>.

75 SINGH, S. P.; SUTTON, R. S. Reinforcement learning with replacing eligibility traces. *Mach. Learn.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 22, n. 1-3, p. 123–158, jan 1996. ISSN 0885-6125. Disponível em: <http://dx.doi.org/10.1007/BF00114726>.

76 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.

77 DERRAC, J.; GARCíA, S.; MOLINA, D.; HERRERA, F. 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. Disponível em: <http://dblp.uni-trier.de/db/journals/swevo/swevo1.html#DerracGMH11>.

78 FARIA, M. P. P.; JULIA, R. M. S.; TOMAZ, L. B. P. Proposal of an Automatic Tool for Evaluating the Quality for Decision-Making on Checkers Player Agents. In: *Proceedings do Encontro Nacional de Inteligência Artificial e Computacional (ENIAC) 2018*. [s.n.], 2018. Disponível em: <https://doi.org/10.5753/eniac.2018.4433>.

79 LU, C. ping P. *Parallel Search of Narrow Game Trees*. Dissertação (Mestrado) — University of Alberta, 1993.

80 SCHAEFFER, J.; CULBERSON, J.; N.TRELOAR; KNIGHT, B.; LU, P.; SZAFRON, D. A World Championship Caliber Checkers Program. *Artificial Intelligence*, v. 53, p. 273–289, February 1992. Disponível em: <https://doi.org/10.1016/0004-3702(92)90074-8>.

81 LESSON 8: Remote Method Invocation. 2018. Disponível em: <https://www.oracle.com/technetwork/java/rmi-141556.html>. Acesso em: 22/10/2018.

82 TANENBAUM, A. S. *Modern Operating Systems*. 3rd. ed. [S.l.]: Prentice Hall Press, 2007. ISBN 9780136006633.

83 THE OpenMP API specification for parallel programming. Https://www.openmp.org/ (Disponível em 19/12/2018.

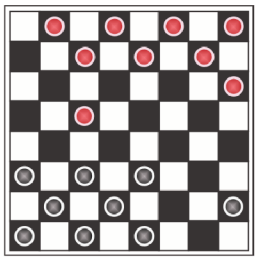
84 THE Message Passing Interface (MPI) standard. Http://www.mcs.anl.gov/research/projects/mpi/ (Disponível em 24/05/2013).

Apêndices

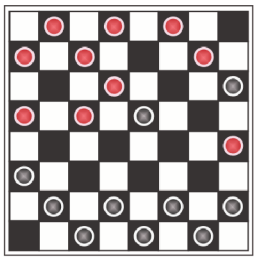
APÊNDICE **A**

Estados dos Experimentos do Adaba

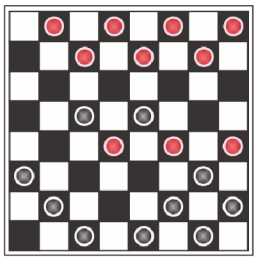
As imagens de estados de tabuleiro relacionados neste apêndice, correspondem àqueles utilizados nos experimentos do ADABA apresentados no capítulo 8. Tais estados foram extraídos da base de testes *Tinsley-Chinook Test Suite 1992* [79].



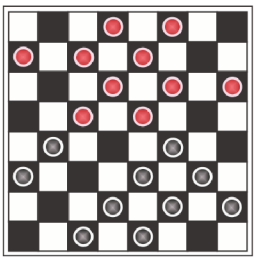
Estado 1



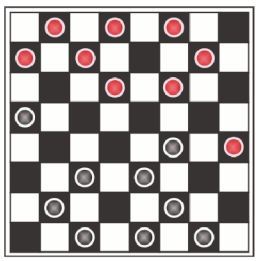
Estado 2



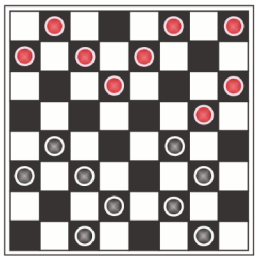
Estado 3



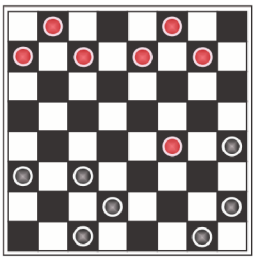
Estado 4



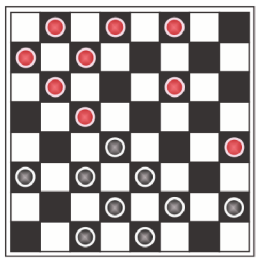
Estado 5



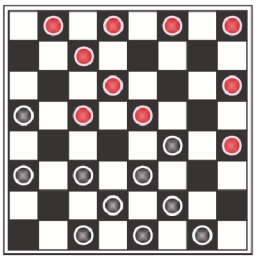
Estado 6



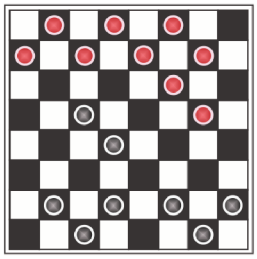
Estado 7



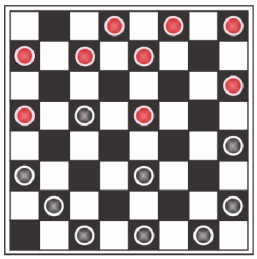
Estado 8



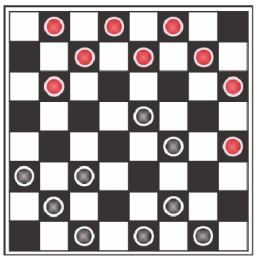
Estado 9



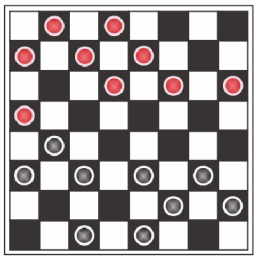
Estado 10



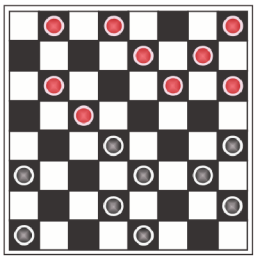
Estado 11



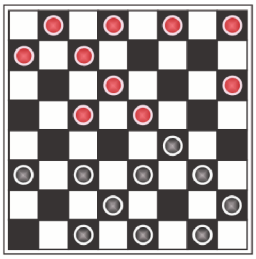
Estado 12



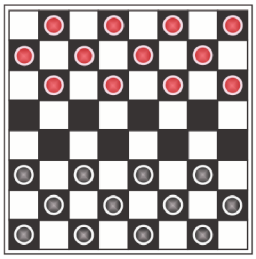
Estado 13



Estado 14



Estado 15



Estado 16