Uma Abordagem para Avaliar o Desempenho de Algoritmos Baseada em Simulações Automáticas de Modelos de Redes de Petri Coloridas Hierárquicas

Clarimundo Machado Moraes Júnior



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

Uberlândia
2017

Clarimundo Machado Moraes Júnior

Uma Abordagem para Avaliar o Desempenho de Algoritmos Baseada em Simulações Automáticas de Modelos de Redes de Petri Coloridas Hierárquicas

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

Área de concentração: Ciência da Computação

Orientador: Profa. Dra. Rita Maria da Silva Julia
Coorientador: Prof. Dr. Stéphane Julia

Uberlândia
2017

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

M827a 2017

Moraes Júnior, Clarimundo Machado, 1969-
Uma abordagem para avaliar o desempenho de algoritmos baseada em simulações automáticas de modelos de Petri coloridas hierárquicas / Clarimundo Machado Moraes Júnior. - 2017.
123 f. : il.

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

1. Computação - Teses. 2. Redes de Petri - Teses. 3. Algoritmos - Teses. 4. Simulação (Computadores) - Teses. I. Julia, Rita Maria da Silva. II. Julia, Stéphane. III. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

CDU: 681.3

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

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da tese de doutorado intitulada "**Uma Abordagem para Avaliar o Desempenho de Algoritmos Baseada em Simulações Automáticas de Modelos de Rede de Petri Coloridas Hierárquicas**" por **Clarimundo Machado Moraes Júnior** como parte dos requisitos exigidos para a obtenção do título de **Doutor em Ciência da Computação.**

Uberlândia, 03 de Março de 2017.

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

Coorientador: _____
Prof. Dr. Stéphane Julia
Universidade Federal de Uberlândia

Banca Examinadora:

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

_____
Prof. Dra. Márcia Aparecida Fernandes
Universidade Federal de Uberlândia

_____
Prof. Dr. José Reinaldo Silva
Universidade de São Paulo

_____
Prof. Dr. João Manuel Portela da Gama
University of Porto - Portugal

Este trabalho é dedicado às pessoas que acreditam que:
" Nunca é tarde quando se quer algo, seja para hoje ou para amanhã.
Tarde foi ontem, quando deixamos de tentar. "

Agradecimentos

Inicialmente, agradeço a Ele que é o único autor e consumador de minha fé, que me faz vitorioso em todas as coisas e que dá o verdadeiro sentido da minha existência: Jesus Cristo de Nazaré. Obrigado pelo amor incondicional e perfeito que o Senhor tem por mim. Glórias eu te dou!

Agradeço aos meus pais, Glória Maria e Clarimundo, que com amor e dedicação me proporcionaram o alicerce afetivo, moral e material para que eu pudesse buscar meu aperfeiçoamento intelectual e profissional: amo vocês.

Agradeço às minhas irmãs, Cláudia e Cristina por todo o amor e convívio familiar que sempre tivemos, me apoiando em meus projetos e comemorando minhas vitórias, obrigado. Ao meu afilhado, Guilherme e sobrinhas Flávia e Ana Luíza, pelos momentos de convívio, amor e alegria, me proporcionando assim incentivos de vida.

Agradeço ao meu saudoso e melhor amigo e irmão, Fábio (Fabim), cujo convívio foi um presente de Deus e estará sempre em meu coração, amo você e até breve.

Agradeço às minhas tias Elite e Edite, que sempre estarão em meu coração e lembrança. Reconheço seus esforços e atos de amor que tiveram para comigo, que Deus as abençoe e obrigado por tudo.

Agradeço a todos os amigos, parentes e colegas de trabalho pelo convívio e contribuição em minha formação como ser humano. Em especial aos primos Donizete, Rose, Marisa e Lucimar, e à amiga Eliane (bibika), amo vocês.

Agradeço, em especial, às duas pessoas que Deus colocou em minha vida acadêmica e que me orientaram, incentivaram, corrigiram e comemoraram cada conquista alcançada em meu aprimoramento intelectual e técnico, os meu orientadores: profa. Dra. Rita Maria da Silva Julia e prof. Dr. Stéphane Julia. Muito obrigado pela dedicação e amizade que demonstraram ter por mim.

Agradeço ao professor Edmilson Rodrigues Pinto do departamento de matemática da UFU, pelo apoio e colaboração técnica realizada na análise dos resultados experimentais parciais e na produção científica.

Por fim, agradeço aos colegas Valquíria, Henrique, Joslaine, Hernani e Luciane pelo

coleguismo, apoio e auxílio técnico em vários momentos da minha pesquisa e aos professores da FACOM que fizeram parte da minha formação acadêmica. Em especial, à amiga profa. Dra. Márcia Aparecida Fernandes pela motivação, contribuição técnica e amizade, à amiga Sara Luzia de Melo pela amizade e incentivo, e ao amigo e secretário da FACOM Erisvaldo Araújo, obrigado.

"O pensamento lógico pode levar você de A a B, mas a imaginação te leva a qualquer parte do Universo."

(Albert Einstein)

Resumo

Dentre as várias abordagens consagradas para análise de desempenho de algoritmos em termos de tempo de execução, destacam-se, por exemplo, a análise assintótica, as técnicas de recorrências e a análise probabilística. Entretanto, há algoritmos que apresentam certas peculiaridades que tornam o uso dessas técnicas puramente matemáticas de avaliação de desempenho inadequadas ou excessivamente árduas. É o caso, por exemplo, de algoritmos cujo tempo de execução pode variar significativamente para um mesmo dado de entrada em função da dinâmica de execução. O mesmo acontece no caso de algoritmos distribuídos em que, dependendo da complexidade da política de distribuição utilizada, a avaliação por meio de métodos analíticos do efeito de um gradual incremento de processadores no seu tempo de execução pode tornar-se impraticável. Em situações como essas, a fim de evitar a alta complexidade matemática envolvida na análise de desempenho desses algoritmos, algumas alternativas baseadas em métodos empíricos ou em modelagem visual vêm sendo adotada pelos pesquisadores. Contudo, ambas alternativas apresentam inconvenientes: no caso dos métodos empíricos, eles requerem a implementação dos algoritmos analisados, o que tem um efeito perverso particularmente no caso dos algoritmos distribuídos, uma vez que eles demandam a aquisição prévia de recursos de hardware dispendiosos de multi-processamento antes mesmo de saber se a proposta de distribuição investigada, de fato, vale a pena. Já as abordagens baseadas em modelos visuais atualmente utilizadas (baseadas em grafos, autômatos e *Unified Modeling Language* - UML) não contam com os recursos dinâmicos necessários para lidar com a avaliação do tempo de execução dos algoritmos. Neste cenário, o presente trabalho propõe uma abordagem visual formal para avaliar o tempo de execução de algoritmos baseada em simulações automáticas de modelos de Redes de Petri Coloridas Hierárquicas (RdPCH) no ambiente gráfico CPN Tools. A abordagem proposta é validada por meio de cálculo dos seguintes parâmetros associados aos algoritmos usados como estudo de caso: a função de complexidade, o tempo de execução real e, no caso dos algoritmos distribuídos, o *speedup* e a eficiência. Foram usados como estudos de caso os seguintes três relevantes algoritmos de busca usados nos agentes da Inteligência Artificial com a finalidade de definir as ações

mais apropriadas que tais agentes devem executar de modo a cumprir seu objetivo, com êxito, em um ambiente em que um oponente tenta minimizar suas chances de sucesso: os algoritmos seriais Minimax e Alpha-Beta; e o algoritmo distribuído PVS. Os resultados obtidos confirmam a correção da abordagem proposta.

Palavras-chave: Rede de Petri Colorida Hierárquica. CPN Tools. Algoritmos. Modelagem. Simulação. Avaliação.

Abstract

Among the various approaches established for analyzing the performance of algorithms in terms of runtime, one can highlight as examples, the asymptotic analysis, recurrence techniques, and probabilistic analysis. However, there are algorithms that present certain peculiarities, which make the use of these purely mathematical performance evaluation techniques inadequate or excessively difficult. This is the case of algorithms for which the runtime varies significantly for the same input data, due to its execution dynamics. The same happens in the case of distributed algorithms, where depending on the complexity of the distribution policy being used, the evaluation by means of analytical methods concerning the effect of a gradual increase in processors on runtime can become impractical. In situations such as these, in order to avoid the high mathematical complexity involved in the performance analysis, researchers have adopted alternatives based on empirical methods or visual modeling. However, both alternatives have drawbacks: in the case of empirical methods, they require the implementation of analyzed algorithms, which has a perverse effect especially in the case of distributed algorithms. This occurs, as they demand the acquisition beforehand of expensive multiprocessing hardware resources before knowing if the distribution proposal under investigation is in fact viable. On the other hand, those approaches based on visual models currently under use (based on graphs, automata and Unified Modeling Language) one notes that does not contain the necessary dynamic resources for dealing with the runtime evaluation of algorithms. In this scenario, the present study proposes a formal visual approach, in order to evaluate the algorithm runtime based on automatic simulations of Hierarchical Colored Petri Net models in the CPN Tools graphic environment. The proposed approach is validated through the calculation of the following parameters associated with the algorithms being used as a case study: the complexity function, real runtime, and in the case of distributed algorithms, speedup and efficiency. The cases studies were based on three relevant search algorithms used in the agents of the Artificial Intelligence, with the aim of defining the most appropriate actions that these agents need to execute in order to fulfil their objective, in an environment where the opponent tries to minimize their chances of success. The algo-

rithms under consideration were the serial algorithms Minimax and Alpha-Beta; and the PVS distributed algorithm. The obtained results confirm the correction of the proposed approach.

Keywords: Hierarchical Colored Petri Net. CPN Tools. Algorithms. Modeling. Simulation. Evaluation.

Lista de ilustrações

Figura 1 – Interação de um Agente com o Ambiente. 36
Figura 2 – Exemplos de Estados do Quebra-Cabeça de 8 Peças. 38
Figura 3 – Árvore de Busca do Quebra-Cabeça de 8 Peças. 38
Figura 4 – Exemplo da Árvore de Busca Expandida pelo Minimax. 39
Figura 5 – Expansão da Árvore de Busca pelo Minimax e pelo Alpha-Beta fail-soft. 42
Figura 6 – Exemplo de Ordenação em uma Árvore de Busca com AI. 43
Figura 7 – Expansão Alpha-Beta em Duas Árvores com Ordenações Diferentes. . . 44
Figura 8 – Exemplo de Expansão do PVS. 46
Figura 9 – Elementos Básicos de uma RdP. 48
Figura 10 – Exemplo do Comportamento Dinâmico de uma RdP. 49
Figura 11 – Modelagem de Restrições. 50
Figura 12 – Modelo de RdP Ordinária para o Sistema de Produção de 3 Peças. . . 53
Figura 13 – Modelo de RdP Ordinária com Política de Disparo. 53
Figura 14 – Evolução Dinâmica do Modelo de RdP Ordinária com Política de Disparo. 54
Figura 15 – Modelo de RdPC para um Sistema de Produção de 3 Peças. 55
Figura 16 – Exemplo da Evolução Dinâmica entre uma Rede e uma Sub-Rede . . . 58
Figura 17 – Exemplo de Comunicação entre Duas Sub-Redes. 59
Figura 18 – Versao 4.0.0 do CPN Tools. 60
Figura 19 – Modelagem de Comandos de Atribuição. 68
Figura 20 – Modelagem de Comandos Condicional e Iterativo. 69
Figura 21 – Exemplo de Função Recursiva. 70
Figura 22 – Modelagem da Chamada Recursiva. 70
Figura 23 – Modelagem do Retorno Recursivo. 71
Figura 24 – Pseudo-Código do Minimax. 72
Figura 25 – Visão Macro do Modelo do Algoritmo Minimax. 72
Figura 26 – Modelando o Algoritmo Minimax. 73
Figura 27 – Exemplos de Declarações no CPN Tools. 75
Figura 28 – Sub-rede VerifyLeaf. 75

Figura 29 – Exemplo de Contabilização dos Tempos de Execução 77
Figura 30 – Modelagem da Sub-rede VerifyMax. 78
Figura 31 – Declaração das Funções de Controle do Comando Iterativo. 79
Figura 32 – Primeiro Conjunto: Tempos de Execução Médios Obtidos Experimentalmente. 81
Figura 33 – Primeiro Conjunto: Comparativo dos Tempos de Execução Experimental e Teórico do Minimax. 82
Figura 34 – Primeiro Conjunto: Comparativo das Funções de Complexidade Experimental e Teórica do Minimax. 83
Figura 35 – Segundo Conjunto: Tempos de Execução Médios Obtidos Experimentalmente. 84
Figura 36 – Segundo Conjunto: Comparativo dos Tempos de Execução Experimental e Teórico do Minimax. 85
Figura 37 – Segundo Conjunto: Comparativo das Funções de Complexidade Experimental e Teórica do Minimax. 86
Figura 38 – Pseudo-Código do Alpha-Beta. 88
Figura 39 – Modelando o Algoritmo Alpha-Beta. 89
Figura 40 – Modelagem dos Tempos de Execução Reais dos Comandos. 89
Figura 41 – Estrutura Parcial da Função de Controle sons(v). 91
Figura 42 – Estrutura Parcial da Função de Controle pruning(v). 92
Figura 43 – Curvas dos Tempos de Execução. 94
Figura 44 – Pilha de Nós a serem Distribuídos. 100
Figura 45 – Lista $l2$ de Processadores em Uso. 101
Figura 46 – Estado Imediatamente Anterior à Alocação do Processador P0 102
Figura 47 – Alocação do Processador P0 103
Figura 48 – Alocação do Processador P1 103
Figura 49 – Speedup do Algoritmo PVS. 106
Figura 50 – Eficiência do Algoritmo PVS. 107
Figura 51 – Comparativo das Curvas: (a) Speedup (b) Eficiência. 107

Lista de tabelas

Tabela 1 – OPERADORES BÁSICOS DA LINGUAGEM STANDARD ML 61
Tabela 2 – COMPARATIVO DE TEMPOS DE EXECUÇÃO 93

Lista de siglas

AI Aprofundamento Iterativo

DTS Dynamic Tree Splitting

IA Inteligência Artificial

MPI Message Passing Interface

PNML Petri Net Markup Language

PVS Principal Variation Splitting

RdPAN Redes de Petri de Alto Nível

RdP Redes de Petri

RdPC Redes de Petri Coloridas

RdPCH Redes de Petri Coloridas Hierárquicas

RdPE Redes de Petri Estocástica

RdPEG Redes de Petri Estocástica Generalizada

RdPLT Redes de Petri Lugar/Transição

RdPPT Redes de Petri Predicado/Transição

RNA Rede Neural Artificial

RS Redes Simétricas

TT Tabela de Transposição

UML Unified Modeling Language

XML Extensible Markup Language

YBWC Young Brothers Wait Concept

Sumário

1 INTRODUÇÃO 25
1.1 Contextualização do Problema e da Solução 25
1.2 Motivação 29
1.3 Objetivos e Desafios da Pesquisa 30
1.4 Hipóteses 31
1.5 Contribuições Científicas 32
1.6 Organização da Tese 33

2 FUNDAMENTAÇÃO TEÓRICA 35
2.1 Agentes Inteligentes 36
2.2 Problemas de Busca 37
2.3 Algoritmo de Busca Serial Minimax 39
2.4 Algoritmo de Busca Serial Alpha-Beta 40
2.4.1 Aprimoramentos da Busca Serial Alpha-Beta 40
2.4.2 Impacto da Ordenação dos Nós-Folha no Algoritmo Alpha-Beta 42
2.5 Algoritmo de Busca Distribuída PVS 45
2.6 Redes de Petri 47
2.6.1 Redes de Petri de Alto Nível 50
2.6.2 Redes de Petri Coloridas 51
2.6.3 Definição Formal 51
2.6.4 Exemplificando a Dinâmica de um Modelo de RdPC 52
2.6.5 Redes de Petri Coloridas Hierárquicas 57
2.6.6 CPN Tools 59
2.6.7 Linguagem *Standard* ML 61

3 TRABALHOS CORRELATOS 63
3.1 Análise de Desempenho de Softwares Baseada em Métodos Empíricos 63

3.2 **Análise de Desempenho de Softwares Baseada em Ferramentas de Modelagem Visual** **64**
3.3 **Análise de Desempenho de Algoritmos Baseada em Redes de Petri** **65**

4 **AVALIAÇÃO DA FUNÇÃO DE COMPLEXIDADE DOS ALGORITMOS POR MEIO DAS RDPCH** **67**
4.1 **Modelagem dos Comandos** **67**
4.1.1 Comando de Atribuição 68
4.1.2 Comandos Condicional e Iterativo 68
4.1.3 Comando Recursivo 69
4.2 **Modelagem do Algoritmo Minimax** **72**
4.2.1 Modelagem da Sub-rede $VerifyLeaf$ 74
4.2.2 Modelagem das Sub-redes $VerifyMax/VerifyMin$ 76
4.3 **Simulações Automáticas do Modelo: Avaliando a Função de Complexidade do Minimax** **79**
4.3.1 Primeiro Conjunto de Simulações 80
4.3.2 Segundo Conjunto de Simulações 81
4.4 **Considerações Relativas ao Capítulo** **82**

5 **AVALIAÇÃO DOS TEMPOS DE EXECUÇÃO REAIS DOS ALGORITMOS POR MEIO DAS RDPCH** **87**
5.1 **Modelagem do Algoritmo Alpha-Beta** **88**
5.1.1 Modelagem dos Tempos de Execução Reais dos Comandos 88
5.1.2 Modelagem do Fator de Ramificação 91
5.1.3 Modelagem da Ocorrência de Podas 92
5.2 **Simulações do Modelo: Avaliando o Tempo de Execução Real do Alpha-Beta** **93**
5.3 **Considerações Relativas ao Capítulo** **94**

6 **CÁLCULO DO SPEEDUP E DA EFICIÊNCIA DOS ALGORITMOS DISTRIBUÍDOS POR MEIO DAS RDPCH** **97**
6.1 **Modelagem do Algoritmo PVS** **97**
6.1.1 Declarações de Tipos Abstratos de Dados 98
6.1.2 Variáveis Locais 99
6.1.3 Variáveis Globais 99
6.1.4 Modelagem do Mecanismo de Alocação de Processadores 101
6.2 **Resultados das Simulações** **104**
6.2.1 Simulação com um Processador 104
6.2.2 Simulação com Diversos Processadores 105

6.3 Considerações Relativas ao Capítulo 107

7 CONCLUSÕES E TRABALHOS FUTUROS 109
7.1 Considerações Finais 109
7.2 Contribuições Científicas 110
7.3 Limitações 111
7.4 Trabalhos Futuros 111

REFERÊNCIAS 113

ANEXOS 117

ANEXO A – MODELAGEM COMPLETA DA RDPCH REFERENTE AO MINIMAX 119

Capítulo 1

Introdução

1.1 Contextualização do Problema e da Solução

São inúmeros os problemas que despertam o interesse da comunidade científica pela relevância teórica e prática cujas soluções ainda não foram encontradas - ou, então, foram sugeridas de maneira pouco eficiente - . No cenário da Computação, para que uma solução proposta por meio de um algoritmo seja eficiente, devem ser considerados dois aspectos: a memória requerida e o tempo de execução gasto. Com a evolução e a desoneração do hardware, em muitas das aplicações o primeiro aspecto tem apresentado um desafio menor quando comparado ao segundo [1]. Dessa forma, a fim de avaliar o nível de qualidade de soluções baseadas em algoritmos, torna-se imprescindível contar com abordagens apropriadas de avaliação do tempo de execução. Dentre as abordagens analíticas consagradas disponíveis para tal fim, destacam-se [1]: a análise assintótica, a qual mede o impacto do aumento da dimensão do dado de entrada no tempo de execução; as técnicas que resolvem as recorrências, usadas para avaliar os algoritmos recursivos (método da substituição, método da árvore de recursão ou método Máster); e a análise probabilística, que usa probabilidades de distribuição dos dados de entrada para estimar o tempo de execução.

Entretanto, particularmente no caso dos algoritmos distribuídos, dependendo da complexidade da política de distribuição utilizada, a avaliação do efeito de um gradual incremento de processadores no seu tempo de execução por meio de métodos analíticos pode tornar-se impraticável (o que ocorre, por exemplo, sempre que o tempo de execução depende de atualizações de valores de parâmetros efetuadas via troca de mensagens [2]).

Em situações como essas, uma alternativa que vem sendo geralmente adotada pelos pesquisadores para evitar a alta complexidade matemática envolvida na análise de desempenho desses algoritmos se baseia no uso de determinados métodos empíricos. De fato, em [3] os autores propuseram um método empírico baseado na complexidade computacional para entender a escalabilidade de programas. Para este caso em particular, entende-se por escalabilidade a capacidade de ganho de desempenho do programa em função do incre-

mento de novas cargas de trabalho [4]. Em [5], foi proposto um método empírico baseado na comparação de desempenho de modelos de programação paralelos para avaliar o *speedup*, a eficiência e a escalabilidade usando um algoritmo de busca paralelo. Contudo, tais alternativas apresentam o seguinte inconveniente: elas requerem a implementação dos algoritmos analisados, o que pode ter um efeito perverso particularmente no caso dos algoritmos distribuídos, uma vez que isso demandaria a aquisição prévia de recursos de *hardware* dispendiosos de multi-processamento antes mesmo de saber se a proposta de distribuição investigada, de fato, vale a pena.

Além desses métodos empíricos, uma outra alternativa para os métodos matemáticos que vem sendo utilizada com êxito são as ferramentas de modelagem visual, a saber: os grafos [1], usados em [6]; os autômatos [7], usados em [8]; e a *Unified Modeling Language* (UML) [9], usada em [10]. Porém, tais técnicas apresentam as seguintes limitações: os grafos e os autômatos possuem um enfoque predominantemente estático na sua forma de representação que compromete o processo de avaliação do tempo de execução por meio de simulações automáticas; já a UML, apesar de contar com alguns recursos técnicos para lidar com modelos dinâmicos - os diagramas de atividades e de sequência [9] - não provê recursos que permitem representar importantes aspectos da dinâmica dos algoritmos distribuídos. Como exemplos de tais recursos, podem-se citar: estruturas de dados, concorrência, alocação de recursos, sincronização, representação dinâmica dos fluxos de dados e funções de distribuição estocásticas.

Neste cenário, o presente trabalho propõe uma abordagem visual formal para avaliar o desempenho de algoritmos em termos de seus tempos de execução, efetuando, para tanto, simulações de modelos de RdPCH no ambiente gráfico CPN Tools. A fim de estruturar a complexidade da execução de tal proposta, este trabalho é desenvolvido ao longo de 3 etapas complementares entre si. Nas duas primeiras, o foco é enfrentar o problema da modelagem e da simulação automática de comandos normalmente utilizados nos algoritmos. Tais etapas são desenvolvidas tendo algoritmos seriais envolvendo um complexo fluxo de controle de comandos como estudo de caso. Na terceira e última etapa, a abordagem introduzida nas duas primeiras etapas para análise de desempenho de algoritmos seriais é enriquecida com a inserção de uma política de distribuição que permite a avaliação de desempenho de algoritmos distribuídos. Os algoritmos escolhidos como estudos de caso representam três relevantes algoritmos usados frequentemente nos programas automáticos resolvedores de problemas da Inteligência Artificial (IA) - chamados agentes - com a finalidade de definir as ações mais apropriadas que eles devem executar de modo a cumprir seu objetivo, com êxito, em um ambiente em que um oponente tenta minimizar suas chances de sucesso (tal como ocorre nos jogadores automáticos de Xadrez e Damas). São eles: os algoritmos seriais Minimax [11] e Alpha-Beta [12]; e o algoritmo distribuído *Principal Variation Splitting* (PVS) [13]. Nesses algoritmos, a cada vez que o agente deve selecionar uma ação, é montada uma árvore de busca em que o nó-raiz representa o estado corrente

do ambiente no qual o agente opera e os demais níveis da árvore representam os estados que podem ser gerados pela execução de ações legais a partir de seus respectivos nós-pai. No caso de agentes para jogos de tabuleiro, por exemplo, esses algoritmos são usados para norteá-los na escolha do movimento apropriado a ser executado em função do tabuleiro corrente. Com isso, a construção da árvore começa com esse tabuleiro no nó-raiz. A partir daí, os demais níveis da árvore são produzidos, recursivamente, pela geração dos sucessores de cada nó da árvore. Cada sucessor, por sua vez, corresponde ao estado de tabuleiro que resultaria da eventual execução de um movimento legal a partir do nó-pai - o que faz com que em agentes jogadores de Damas e Xadrez, por exemplo, as árvores de busca geradas por tais algoritmos tenham um fator de ramificação extremamente variável (não-uniforme) [14] - . Quanto mais profunda for a árvore de busca, mais abrangente será a visão futura de jogo do agente e, consequentemente, melhor será a qualidade do movimento selecionado pelo algoritmo para ser executado a partir do tabuleiro representado na raiz da árvore [15]. Obviamente, a geração de árvores mais profundas requer maior tempo de processamento. Entretanto, geralmente os jogadores (inclusive automáticos) devem obedecer a um limite de tempo em suas tomadas de decisão. Logo, no caso dos agentes jogadores, quanto mais rápido for o algoritmo de busca, maior profundidade atingirão as árvores de busca produzidas para a escolha do movimentos e, consequentemente, maior será a qualidade dos movimentos por eles executados.

Saliente-se a seguinte relação entre os três algoritmos aqui escolhidos como estudo de caso: todos produzem resultados equivalentes para o mesmo dado de entrada tratado, ou seja, considerando, por exemplo, o caso de agentes jogadores, todos selecionarão um mesmo movimento para um mesmo tabuleiro de jogo. Isso ocorre porque, na realidade, tanto o Alpha-Beta quanto o PVS são versões aprimoradas do Minimax. Mais especificamente, o algoritmo serial Alpha-Beta torna o Minimax muito mais rápido por operar com janelas que permitem a eliminação (podas *Alpha* e *Beta*) dos ramos da árvore a que a solução procurada não pertence [15]. Por outro lado, o PVS reduz o tempo de processamento da busca por meio de multi-processamento. Convém salientar que há versões alternativas do PVS que contemplam ou não as podas *Alpha* e *Beta* [16]. Neste trabalho, a fim de concentrar os esforços iniciais unicamente na resolução de modelagem e simulação da dinâmica distribuída, optou-se por uma versão do PVS que não efetua as referidas podas. A inclusão das podas está proposta como trabalho futuro. Os três algoritmos usados como estudos de caso foram cuidadosamente escolhidos por apresentarem características técnicas bastante apropriadas para justificar a pertinência da metodologia proposta e das ferramentas utilizadas. De fato, ambos os algoritmos seriais usados são fortemente lastreados em ninhos de comandos condicionais e de repetição - incluindo chamadas e retornos recursivos [15] - que demandam uma estrutura de fluxo de comando com sofisticadas peculiaridades, tanto na modelagem, quanto na simulação da dinâmica desses. Além disso, conforme será visto, o algoritmo serial Alpha-Beta apresenta um comportamento peculiar

no tempo de execução, uma vez que o tempo que ele gasta para processar um mesmo dado pode variar em função da ordem em que são gerados os nós sucessores de cada nó explorado da árvore. Isso ocorre em virtude do impacto que a ordenação da árvore ocasiona no número possível de podas *Alpha* e *Beta* que o algoritmo pode efetuar [17]. Quanto ao algoritmo distribuído usado como estudo de caso para testar a adequação da metodologia proposta ao cálculo do impacto que a variação do número de processadores provoca no tempo de execução do algoritmo, conforme citado anteriormente, foi escolhida uma versão do PVS que distribui o Minimax sem considerar as podas. Essa escolha foi feita em virtude de tal algoritmo representar uma abordagem de distribuição que, apesar de modesta - por não incluir, por exemplo, as trocas de mensagens entre os processadores requeridas para comunicar as eventuais atualizações das janelas de busca que ocorrem na segunda versão do PVS - envolve a dinâmica básica de distribuição de tarefas entre multiprocessadores. Tal estratégia proporciona, assim, uma maneira mais didática e simples de efetuar uma primeira abordagem útil e representativa do árduo problema de estimar a eficiência e o *speedup* de algoritmos distribuídos prescindindo da implementação dos mesmos.

Os resultados da abordagem aqui proposta, ou seja, - avaliar o desempenho de algoritmos por meio de uma abordagem formal e visual baseada em simulações automáticas dos modelos de Redes de Petri Coloridas Hierárquicas (RdPCH) correspondentes a tais algoritmos no ambiente CPN tools - , são validados por meio de testes comparativos que são efetuados com relação aos seguintes parâmetros associados aos algoritmos usados como estudo de caso: a função de complexidade, o tempo de execução real e, no caso dos algoritmos distribuídos, o *speedup* e a eficiência. Neste contexto, os três algoritmos citados serão usados para testar a abordagem proposta da seguinte maneira: o Minimax será usado para avaliar a adequação da metodologia visual ao cálculo de função de complexidade de algoritmos. Tal avaliação será feita comparando-se os resultados obtidos pela aplicação da técnica proposta com os resultados obtidos pelo método analítico tradicional [1]. A pertinência da aplicação da abordagem proposta à estimativa do tempo de execução real de algoritmos será avaliada por meio do algoritmo Alpha-Beta. Nesse caso, a avaliação da adequação da técnica será executada pela comparação entre os valores obtidos pela abordagem proposta e os valores médios obtidos em execuções reais de um programa correspondente ao algoritmo Alpha-Beta. Quanto à avaliação da adequação da metodologia proposta à tarefa de estimar o impacto que variações no número de processadores ocasionam no tempo de execução dos algoritmos distribuídos, ou seja, quanto à avaliação do cálculo do *speedup* e da eficiência de algoritmos distribuídos efetuado pela abordagem proposta, ela será feita por meio da aplicação de tal abordagem à versão do algoritmo PVS usando o Minimax. Para lidar com tal problema, o autor precisou implementar nos modelos criados uma política de distribuição capaz de representar a dinâmica de distribuição de tarefas entre os processadores adotada pela algoritmo.

Finalmente, é importante ressaltar que a abordagem proposta foi também avaliada para situações em que os algoritmos são aplicados a problemas que envolvem dados não-uniformes, tal como ocorre em jogos como Damas em que o fator de ramificação da árvore de busca é extremamente desequilibrado [14] (devido à forte variação na quantidade dos movimentos legais possíveis ao longo de um jogo). Em tais casos, as funções de controle implementadas no modelo devem estar aptas a gerar os parâmetros referentes aos dados simulados por meio de probabilidades e de funções de distribuição estocástica.

1.2 Motivação

Uma ferramenta bastante promissora no campo da modelagem de sistemas dinâmicos, são as Redes de Petri (RdP) [18] que, além de possuírem uma abordagem matemática, gráfica e formal, permitem que se façam a modelagem, a análise, o controle e a supervisão de sistemas dinâmicos. Outra potencialidade das RdP é a capacidade de modelar aspectos de sincronismo, alocação de recursos e concorrência - essenciais no caso de tratamento de algoritmos distribuídos - que não podem ser representados em outras ferramentas como os autômatos.

Convém salientar que outros trabalhos tais como [19] e [20] também se baseiam em modelagem de RdP para representar o controle de fluxos de comando em algoritmos seriais. Contudo, essas abordagens não provêem os recursos necessários para lidar com os seguintes desafios: controle de fluxos de comandos que sofrem sucessivos desvios, tais como os condicionais, os iterativos e as chamadas/retornos recursivos (requerido na modelagem de algoritmos como o Minimax e o Alpha-Beta); e simulação automática dos modelos construídos requerida para o cálculo do tempo de execução. Além disso, em [21], [22] e [23], os autores também usam modelagem de RdP para tratar aspectos de processos distribuídos, contudo em nenhum desses trabalhos é implementada uma política de distribuição e de controle em algoritmos distribuídos que possibilite as simulações automáticas a partir das quais é feito o cálculo do *speedup* e da eficiência desses algoritmos. É importante destacar também que, diferentemente da presente proposta, nenhum desses trabalhos correlatos citados acima lida com dados não-uniformes.

Quando o objetivo é modelar algoritmos distribuídos, dentre os diversos modelos de RdP existentes, as Redes de Petri Coloridas (RdPC) se destacam como ferramenta apropriada, uma vez que, além de possuírem todas as potencialidades inerentes às RdP, apresentam duas características que justificam ainda mais o seu uso: permitem agregar informações (cores) em suas fichas, tornando seus modelos mais compactos estruturalmente; e possibilitam que se associem aos modelos os recursos obtidos com o paradigma de programação das linguagens funcionais, fato que permite a definição e manipulação de tipos de dados complexos. Outro aspecto relevante a ser considerado na modelagem de algoritmos é o fato de que, frequentemente, eles são estratégica e convenientemente concebidos sob

uma forma estruturada em que as partes que os compõem apresentam funcionalidades distintas e específicas. Nesses casos, tal como é feito na técnica de modularização em programação, torna-se oportuno decompor o modelo global em partes menores que representem tais funcionalidades. Tal objetivo pode ser atingido com o uso das RdPCH, as quais correspondem às RdPC acrescidas de técnicas de hierarquização. Tais técnicas permitem a criação de sub-redes e o estabelecimento de hierarquia e de comunicação entre elas, o que possibilita uma dinâmica em que as fichas podem fluir ao longo do modelo de uma maneira correta.

Para que a avaliação de um algoritmo seja feita com sucesso, além da modelagem apropriada, é preciso que o ambiente de simulação ofereça todos os recursos necessários para que todos os cenários produzidos pelo algoritmo sejam passíveis de serem simulados. Neste contexto, o ambiente do CPN Tools é ideal para desempenhar esse papel, pois apresenta diversos recursos que permitem, por exemplo, a edição, manipulação, simulação e monitoramento de modelos de RdPCH. Além disso, o CPN Tools é uma ferramenta de domínio público.

Com base nas argumentações apresentadas, este trabalho de tese propõe uma abordagem baseada na simulação automática de modelos formais de RdPCH construídos no ambiente CPN tools, com a finalidade de avaliar o desempenho de algoritmos seriais e distribuídos em termos do tempo de execução. Para tanto, o presente trabalho foi conduzido ao longo de 3 etapas, cada uma delas concebida de forma a atingir os objetivos descritos na próxima seção.

1.3 Objetivos e Desafios da Pesquisa

O objetivo geral desta tese é propiciar uma abordagem para avaliar o desempenho de algoritmos seriais e distribuídos baseada em simulações automáticas de modelos formais, abordagem, esta, que prescinda da implementação dos algoritmos analisados. Tal abordagem deverá ser testada, inclusive, em situações em que os algoritmos usados como estudo de caso se aplicam a problemas que lidam com dados não-uniformes. Para atingir o objetivo geral, foram traçados três objetivos específicos, a saber:

- Calcular a função de complexidade do algoritmo Minimax com base na técnica proposta. A escolha do Minimax se deve à dificuldade de se representar o controle de seu fluxo de comandos, o qual envolve comandos condicionais, repetitivos e chamadas/retornos recursivos. A avaliação da adequação da técnica proposta será feita comparando-se os resultados obtidos por ela com a conhecida função de complexidade do Minimax calculada pelo método analítico tradicional;

- Modelar o algoritmo Alpha-Beta a partir da técnica proposta. Tal escolha se fundamenta na necessidade de se representar a ocorrência de podas que eventualmente

aparecem durante a exploração da árvore de busca. Uma vez que a ocorrência das podas depende da ordenação dos ramos da árvore, o modelo precisa lidar com informações estocásticas, conferindo um desafio maior na modelagem. A avaliação da eficácia da abordagem proposta será feita por meio de comparação entre os resultados obtidos por ela e o tempo de execução real de um programa em que se implementa o Alpha-Beta;

- Calcular o *speedup* e a eficiência do algoritmo distribuído PVS por meio da técnica proposta. A validação dos resultados obtidos pela técnica será feita por meio de uma comparação entre eles e os resultados provenientes de análises do comportamento do algoritmo realizadas em outros trabalhos.

É importante salientar que os três objetivos específicos foram traçados de modo que o cumprimento de um servisse como base para a elaboração dos demais. Essa estratégia permitiu estruturar a complexidade e a dificuldade de se lidar com algoritmos distribuídos. De fato, a modelagem do Minimax permitiu representar tanto a estrutura de diversos comandos quanto de seus fluxos de controle. Além disso, a simulação automática do modelo gerado mostrou a viabilidade de simular a dinâmica real dos algoritmos de um modo geral. Dessa forma, o cumprimento do primeiro objetivo serviu de base para realizar os outros dois. Já a modelagem do Alpha-Beta serviu de base para uma futura proposta de modelar versões distribuídas de algoritmos que produzem tempos de execução variáveis para um mesmo dado de entrada.

1.4 Hipóteses

Com base nos objetivos definidos na seção 1.3, duas hipóteses foram levantadas:

- Mesmos os algoritmos estruturalmente complexos, quer sejam seriais ou distribuídos, podem ser modelados por meio de RdPCH, sendo que os modelos construídos podem ser simulados automaticamente usando o ambiente CPN Tools;

- As simulações automáticas dos modelos construídos para os algoritmos podem ser utilizadas para calcular a função de complexidade, o tempo de execução real e, no caso dos algoritmos distribuídos, a eficiência e o *speedup* dos mesmos. Tais cálculos podem ser feitos, inclusive, para situações em que os algoritmos lidam com problemas que envolvem dados não-uniformes. Os resultados desses cálculos podem ser validados por meio de comparação com outros obtidos por métodos analíticos, pela implementação desses algoritmos ou por análise lógica do comportamento dos mesmos.

1.5 Contribuições Científicas

As principais contribuições científicas desta tese são:

- Uma abordagem formal e visual para análise de desempenho de algoritmos seriais e distribuídos baseada em simulações automáticas de modelos construídos para representá-los. Tal abordagem está apta a lidar com algoritmos que envolvem um dinâmico fluxo de comando ocasionado por comandos condicionais e de repetição e por chamadas/retornos recursivos, além de se aplicar também a situações em que os algoritmos operam com dados não-uniformes. O autor acredita que a abordagem proposta torna-se bastante útil em situações que envolvem algoritmos cujo tempo de execução dificilmente seria passível de ser calculado por métodos analíticos tradicionais. Além disso, a presente abordagem representa uma alternativa bastante útil para calcular o *speedup* e a eficiência de algoritmos distribuídos sem que seja necessário implementá-los;

- O autor acredita que a abordagem proposta representa uma ferramenta importante para análise de alternativas para melhorar o desempenho dos algoritmos distribuídos. Mais especificamente, a aplicação das propriedades das RdP aos modelos construídos pode auxiliar a detectar novas possibilidades de re-distribuição de tarefas entre os processadores disponíveis, reduzindo o ócio dos mesmos;

- Artigo completo em conferência referente ao primeiro objetivo da seção 1.3: "JÚNIOR, C. M. M.; JULIA, R. M. S.; JULIA, S. *Modeling Recursive Search Algorithms By Means of Hierarchical Colored Petri Nets and CPN Tools.* In: 12th International Conference on Information Technology: New Generations on (ITNG), p. 788-791, Las Vegas, USA, 2015." (Ciência da Computação: Qualis B1);

- Artigo completo em conferência atual referente ao primeiro objetivo da seção 1.3: "JÚNIOR, C. M. M.; JULIA, R. M. S.; JULIA, S.; SILVA, L. F. *A New Approach to Evaluate the Complexity Function of Algorithms Based on Simulations of Hierarchical Colored Petri Net Models.* In: 14th International Conference on Information Technology: New Generations on (ITNG), Las Vegas, USA, April 2017." (Ciência da Computação: Qualis B1);

- Submissão atual de artigo completo em conferência referente ao segundo objetivo da seção 1.3: "JÚNIOR, C. M. M.; JULIA, R. M. S.; JULIA, S.; PINTO, E. R.; SILVA, L. F. *Estimating the Runtime of Algorithms With Non Predictive Behavior Through a Timed Hierarchical Colored Petri Net-based Model.* In: 38th International Conference on Application and Theory of Petri Nets and Concurrency, and the 17th International Conference on Application of Concurrency to System Design (ACSD/PN), Zaragoza, Spain, June 2017." (Ciência da Computação: Qualis B1);

- Submissão atual de artigo em revista referente ao terceiro objetivo da seção 1.3: "JÚNIOR, C. M. M.; JULIA, R. M. S.; JULIA, S.; SILVA, L. F. *A New Approach to Evaluate the Speedup and the Eciency of Distributed Algorithms Based on Automatic Simulations of Models.* Journal of Performance Evaluation (PEVA), Elsevier." (Ciência da Computação: Qualis A2).

1.6 Organização da Tese

Essa tese está estruturada da seguinte forma:

- O capítulo 1 é a introdução do trabalho;
- O capítulo 2 consiste na fundamentação teórica;
- O capítulo 3 evidencia o estado da arte.
- O capítulo 4 detalha a modelagem do algoritmo de busca serial Minimax, bem como, a simulação automática de diversos cenários para a obtenção da sua função de complexidade;
- O capítulo 5 detalha a modelagem e simulação automática do algoritmo de busca serial Alpha-Beta de modo a calcular seu tempo de execução real;
- O capítulo 6 detalha a modelagem da política de distribuição do algoritmo de busca distribuída PVS, simulando automaticamente diversos cenários para a obtenção dos seus *speedup* e de suas eficiências;
- O capítulo 7 explicita as considerações finais apontadas com base nos resultados obtidos, bem como, as contribuições científicas conquistadas, as limitações encontradas e as propostas de trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Na área da computação, o desejo de querer solucionar problemas cada vez mais complexos fez com que os pesquisadores propusessem algoritmos cada vez mais sofisticados. Um exemplo disso são os chamados jogos de tabuleiro investigados no campo da IA. Tais problemas apresentam como desafio principal a proposição de agentes jogadores capazes de explorar grandes espaços de busca por meio de algoritmos de busca num menor intervalo de tempo possível e escolher assim, a melhor opção de jogada. A medida que os algoritmos de busca foram sendo propostos, para que produzissem resultados cada vez mais eficazes, suas complexidades estruturais e/ou comportamentais aumentaram consideravelmente. Por exemplo, quando foi proposto o algoritmo de busca Minimax [11], este produziu resultados bem satisfatórios na época. Posteriormente, aprimorou-se a busca Minimax com a inclusão das podas culminando no algoritmo Alpha-Beta [12], propiciando melhores resultados por parte dos agentes jogadores que o utilizaram. Mais tarde, foram propostas versões distribuídas do Alpha-Beta, como por exemplo o PVS [13], adicionando-se uma política de distribuição dos processadores envolvidos e fazendo com que os tempos de execução produzidos fossem bem menores quando comparados com a versão serial do Alpha-Beta.

Com o aumento da sofisticação desses algoritmos, é desejável que se avalie minuciosamente seus comportamentos de modo a entender melhor seus comportamentos e com isto, abrir a possibilidade de se propor versões melhoradas dos mesmos ou de novos algoritmos. O uso de ferramentas puramente matemáticas para essas avaliações nem sempre é uma tarefa fácil, sugerindo assim o uso de abordagens empíricas, de modo especial, de ferramentas baseadas em simulações automáticas de modelos que representem esses algoritmos, sem a necessidades de implementá-los. As RdP são ferramentas de modelagem gráfica e formal que apresentam um enfoque dinâmico e permitem a simulação automática dos modelos. Além disso, as RdP [18] conseguem representar aspectos importantes como alocação de recursos, sincronização e concorrência que estão presentes, por exemlo, nos algoritmos distribuídos. Diversos modelos de RdP já foram propostos, dentre eles estão as RdPCH [23]. Tais modelos são extensões das RdPC cujo destaque está na ob-

tenção de uma modelagem mais compacta. Além disso, as RdPCH apresentam o recurso da hierarquização das sub-redes envolvidas. É importante destacar que as RdPCH podem ser produzidas manualmente e simuladas automaticamente no ambiente gráfico CPN Tools [24] que possui diversos recursos e é uma ferramenta de domínio público.

2.1 Agentes Inteligentes

Um agente é tudo que possui a capacidade de perceber seu ambiente por meio de sensores e de agir sobre esse ambiente por intermédio de atuadores [11]. A Figura 1 mostra que um agente deve possuir um mecanismo que relaciona a percepção do ambiente com a ação produzida.

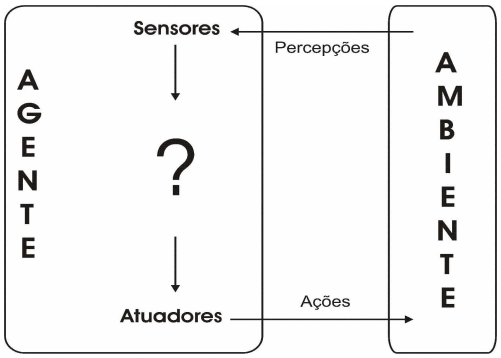


Figura 1 – Interação de um Agente com o Ambiente.

A implementação desse mecanismo, conhecida como "Programa de Agente", determina o tipo de agente que será produzido. Assim, o agente pode ser [11]:

- ❏ reativo simples - seleciona ações com base apenas na percepção atual;
- ❏ reativo baseado em modelos - possui estados internos, além da percepção atual, que ajudam na escolha das ações;
- ❏ baseado em objetivos - possui estados internos que aliados a um objetivo, definem a ação a ser tomada;
- ❏ baseado na utilidade - por meio de um mapeamento e avaliando estados internos, valores são gerados definindo o quão útil será a ação escolhida;

Os quatro tipos de agentes citados anteriormente, não conseguem lidar com situações em que a experiência deve ser levada em conta na escolha das ações a serem tomadas. Para isso, uma boa opção é incluir no mecanismo do agente a capacidade de aprendizagem. Esse aprimoramento é construído por meio de um algoritmo responsável por aprender com situações novas produzindo as melhores ações. Os agentes com aprendizagem são comumente conhecimentos como "Agentes Inteligentes".

Na área da IA, mais especificamente em aplicações de jogos de tabuleiro, o número de possibilidades de ações a serem feitas costuma ser bem elevado. Isso fez com que o uso de um agente jogador em tais problemas configurasse uma linha de pesquisa bastante interessante uma vez que os pesquisadores têm sugerido Programas de Agente cada vez mais sofisticados por meio das combinações de várias técnicas. Dentre essas técnicas, vale destacar as técnicas de buscas que são usadas para auxiliar na tomada de decisão para encontrar a melhor jogada por parte do agente jogador, de modo a obter a vitória sobre seu oponente.

2.2 Problemas de Busca

O que diferencia uma técnica de busca da outra são os critérios usados por elas para escolher uma determinada ação. Quando nenhuma informação específica do problema a ser resolvido é considerada nessa escolha, a técnica é chamada de busca não-informada, que costuma produzir efeitos ruins no desempenho das jogadas do agente jogador. Porém, os agentes jogadores baseados em objetivos tem apresentado bons resultados, como por exemplo, em jogos de tabuleiros. Isso se deve ao fato de que, em aplicações de jogos, o objetivo do agente jogador é sempre bem definido, ou seja, alcançar a vitória sobre seu oponente. Esse objetivo se traduz em instâncias com características bem particulares de cada tipo de jogo, por exemplo, em jogos de damas a instância a ser alcançada é uma configuração do tabuleiro em que não haja nenhuma peça do oponente sobre ele. Cada instância é chamada de estado e o conjunto de todas as instâncias é dito **espaço de estados**.

A Figura 2 mostra o problema do quebra-cabeça de 8 peças com tabuleiro 3x3 (9 posições distintas). Nota-se que há 8 peças numeradas de 1 a 8 posicionadas no quebra-cabeça e que uma posição está vazia. Na Figura 2, o estado inicial é uma instância do quebra-cabeça na qual as 8 peças estão posicionadas aleatoriamente. O problema consiste em movimentar as peças a partir do estado inicial de modo a obter o estado objetivo pré-definido, conforme mostra a Figura 2. Para isso as peças adjacentes à posição vazia, podem ser movimentadas horizontalmente ou verticalmente para essa posição. Por exemplo, a partir do estado inicial mostrado na Figura 2, as possibilidades de ação são: mover peça 5 verticalmente para baixo, mover peça 1 verticalmente para cima, mover peça 7 horizontalmente para a direita ou mover peça 8 horizontalmente para a esquerda. Ao realizar uma dessas ações, um novo estado é produzido, podendo levar ou não à obtenção do estado objetivo. Para o quebra-cabeça de 8 peças o espaço de estados possui 181.444 estados acessíveis.

Computacionalmente, para propor um agente jogador que encontre o estado objetivo do jogo, é necessário representar as sequências de estados acessíveis por meio de uma estrutura de dados. Para isso, são usadas as chamadas **árvores de busca** [11] que são

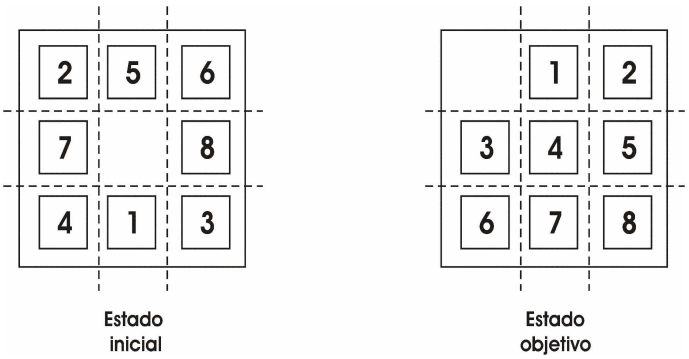


Figura 2 – Exemplos de Estados do Quebra-Cabeça de 8 Peças.

caracterizadas por possuírem um nó-raiz que contém o estado atual do problema e nós sucessores desse nó-raiz contendo outros estados distintos. No exemplo da Figura 3, a árvore de busca apresenta um nó-raiz contendo o estado inicial do quebra-cabeça de 8 peças. Além disso, o nó-raiz possui 4 nós sucessores chamados de nós-filho que representam os estados obtidos a partir dos movimentos das peças mencionados anteriormente. Considerando todas as sequências dos estados acessíveis, tem-se o chamado **espaço de busca**. Para o quebra-cabeça de 8 peças o espaço de busca é infinito.

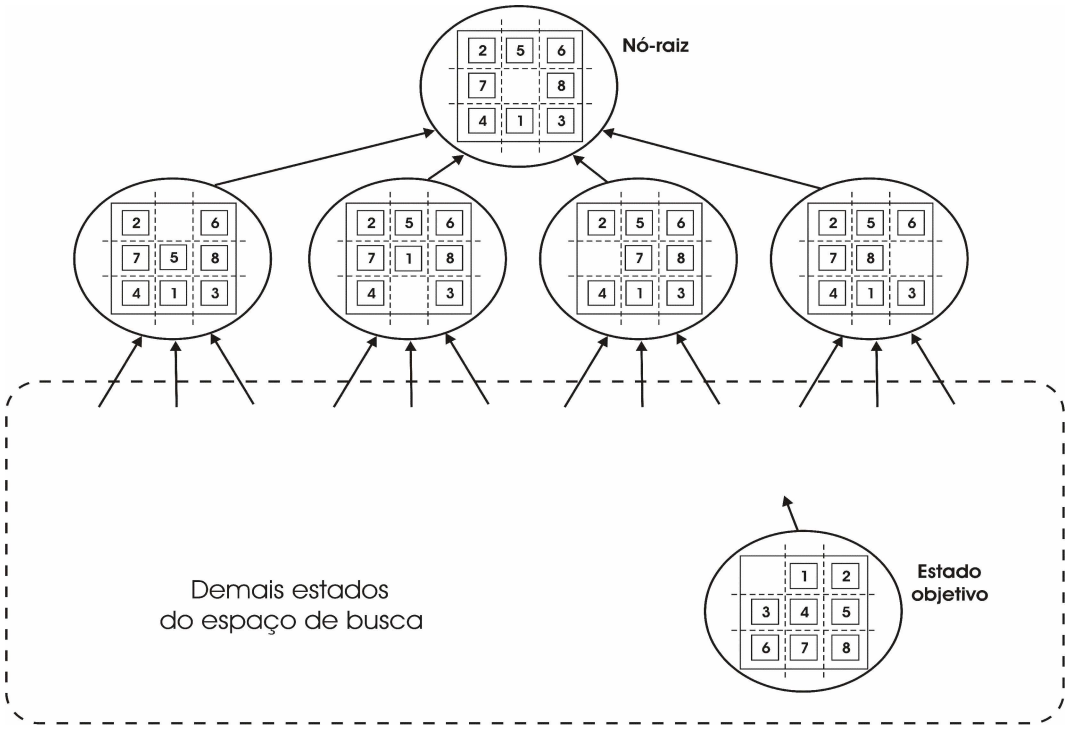


Figura 3 – Árvore de Busca do Quebra-Cabeça de 8 Peças.

Uma vez que os espaços de busca em problemas como jogos de tabuleiros são muito grandes ou infinitos, a proposição de algoritmos de busca mais eficientes tem sido o alvo de muitos pesquisadores da área. O fato é que além de ter que encontrar o estado objetivo do jogo (instância representando a vitória do agente jogador), o algoritmo deve fazê-lo

respeitando um limite de tempo aceitável. Portanto, diversos algoritmos de busca têm sido propostos diferenciando-se entre si, basicamente, pelos aspectos do tempo de execução gasto e da memória requerida. Vale destacar que em muitas aplicações, com a desoneração do hardware, o primeiro aspecto tem apresentado um desafio maior que o segundo.

2.3 Algoritmo de Busca Serial Minimax

O Minimax [11] é um algoritmo de busca serial, comumente usado nos agentes jogadores que processam informações perfeitas - cada jogador tem a sua vez de jogar permitindo assim que cada um deles saiba o estado atual do jogo. Explorando uma árvore de busca, o Minimax tem como objetivo escolher o movimento (ação) apropriado a partir do estado atual do jogo (nó-raiz). Após encontrar os valores calculados nos nós-folha, tais valores são propagados para os níveis anteriores na árvore que, alternadamente, representam o nível de maximização para o agente e o nível de minimização para o oponente, de modo a encontrar a solução. A Figura 4 mostra a expansão de uma árvore de busca feita pelo algoritmo Minimax em um jogo de tabuleiro, no qual cada nó representa um estado a ser encontrado a partir do nó-raiz.

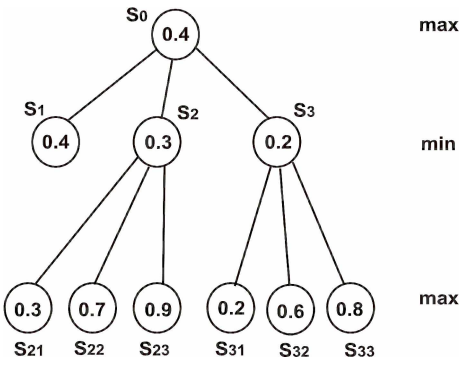


Figura 4 – Exemplo da Árvore de Busca Expandida pelo Minimax.

A busca é feita em profundidade [11], sempre iniciando nos nós mais à esquerda na árvore. Quando um nó-folha é encontrado, uma avaliação é realizada, de acordo com uma heurística pré-estabelecida. De posse desse valor calculado, o algoritmo decide se irá propagá-lo ou não para os níveis anteriores. Analisando a Figura 4, quando a busca na árvore é iniciada, o nó-folha mais à esquerda (S_1) tem uma avaliação igual a 0.4. Como o nó do nível anterior a ele (S_0) ainda não recebeu nenhuma propagação de volta de algum valor, então 0.4 é atribuído a S_0. Mais uma vez, explorando em profundidade para a sub-árvore mais à esquerda (que ainda não foi explorada), os próximos nós-folhas são avaliados com $S_{21} = 0.3$, $S_{22} = 0.7$ e $S_{23} = 0.9$. Como o nó do nível anterior (S_2) se encontra em um nível de minimização, o valor a ser propagado de volta para S_2 será 0.3 (menor valor atual). Considerando que o nó-raiz já tem o valor 0.4 e que seu nível é de maximização, o máximo valor de 0.4 foi mantido, sem ocorrer a propagação de volta do

valor de S_2 para S_0. Após explorar toda a árvore de busca, o nó-raiz terá um valor que vai permitir decidir qual sub-árvore irá representar a melhor escolha, isto é, o nó filho do nó-raiz com uma avaliação igual a ele, será a melhor opção. No exemplo mostrado, a melhor opção é o nó S_1. Considerando o comportamento do Minimax, é importante destacar que:

- a sua função teórica da complexidade de tempo de execução é a mesma para a busca em profundidade, a saber $O(b^m)$ [11], em que b é o fator de ramificação (número de nós-filho) médio da árvore e m é a sua profundidade máxima;
- o seu tempo de execução em uma mesma árvore de busca é sempre o mesmo (explora toda a árvore de busca) independente da ordenação dos seus nós-folha;
- em jogos de tabuleiro, ele apresenta a desvantagem de gerar e avaliar nós que não são relevantes na busca, ou seja, que não representam opções apropriadas de movimento no jogo. Há uma proposta do Minimax que resolve esta desvantagem, conhecida como algoritmo Alpha-Beta e que será detalhado na seção 2.4.

Em razão de o desempenho do algoritmo Minimax já ter sido analisado teoricamente, este algoritmo constitui, por si só, uma boa opção de estudo de caso para novas abordagens de análise de algoritmos. Desse modo, na etapa de validação é possível comparar seus resultados já conhecidos pela comunidade cientifica com os resultados obtidos por meio da nova abordagem sugerida.

Ainda que seu processo de busca seja feito sempre no pior caso (exploração de toda árvore), o Minimax foi uma boa escolha para compor o módulo de busca em agentes jogadores, como proposto em [25] e [26].

2.4 Algoritmo de Busca Serial Alpha-Beta

Com o intuito de melhorar o desempenho do algoritmo Minimax, o algoritmo de busca serial Alpha-Beta [12] propõe o uso de uma janela de observação que tende a estreitar-se em direção da solução, segundo uma heurística específica. Este estreitamento permite que haja a exclusão de nós irrelevantes conhecida como podas *Alpha* e *Beta* [12]. Além do ganho obtido por meio das podas na busca Alpha-Beta, alguns aprimoramentos já foram sugeridos em sua versão original e serão destacadas na seção 2.4.1.

2.4.1 Aprimoramentos da Busca Serial Alpha-Beta

Para acoplar as vantagens da estratégia de busca em largura (completude e valor ótimo) e a estratégia de busca em profundidade limitada (complexidade de espaço de

$O(bm)$ [11], em que b é o fator de ramificação e m é a profundidade máxima), a estratégia de Aprofundamento Iterativo (AI) é frequentemente usada na implementação dos algoritmos de busca para agentes jogadores [27]. Essa estratégia efetua sucessivas buscas em profundidade limitada, sendo que tal limite é gradualmente incrementado a partir da raiz. Uma grande desvantagem do AI é a repetição do processamento de estados nos níveis menores da árvore de busca. Uma solução é usar uma Tabela de Transposição (TT), a qual carrega os dados relevantes dos nós já visitados durante a busca, evitando assim avaliações redundantes. A combinação do algoritmo de busca serial Alpha-Beta + TT + AI melhora significativamente o desempenho do algoritmo de busca para o agente jogador.

Em [27], os autores mostraram que a versão original *hard-soft* do algoritmo Alpha-Beta, proposto em [12], não pode ser usada juntamente com uma TT, pois isto resultaria em um erro na escolha para os melhores movimentos (que seriam diferentes dos movimentos indicados pelo Minimax). Assim sendo, com o agente jogador VisionDraughts, propôs-se o uso de uma versão *fail-soft* do Alpha-Beta. As Figuras 5(a) e 5(b) ilustram uma expansão Minimax e sua expansão Alpha-Beta *fail-soft* correspondente, respectivamente. Nessas árvores de busca, os valores dos nós-folha são calculados por uma função de avaliação apropriada e os valores dos nós restantes são obtidos pela propagação de volta dos nós-folha por meio de uma estratégia Minimax. Então, o movimento cuja execução a partir do nó-raiz produz o nó-filho (nó gerado a partir do nó-raiz) com a melhor avaliação é escolhido e executado. Na árvore da versão *fail-soft* da Figura 5(b), a poda ocorre da seguinte forma:

- a busca em profundidade é feita na direção da esquerda para a direita;
- os valores iniciais mínimo (*Alpha*) e máximo (*Beta*) da janela de busca são, respectivamente, $-\infty$ e $+\infty$;
- o estreitamento da janela pode ocorrer durante a exploração na árvore somente nos retornos aos níveis anteriores. Se o retorno é em um nó minimizador, o estreitamento é no sentido de diminuir o valor de *Beta*. Porém, se o retorno é em um nó maximizador, o estreitamento é no sentido de aumentar o valor de *Alpha*;
- a avaliação do nó-filho de um nó minimizador n pode ser interrompida no momento que uma predição calculada é menor que o valor *Alpha* (poda alpha);
- a avaliação de um nó-filho de um nó maximizador n pode ser interrompida no momento que uma predição calculada é maior que o valor *Beta* (poda beta).

Vale destacar que, no processamento de um nó, as podas *Alpha* e *Beta* ocorrem somente após a conclusão de sua ramificação mais à esquerda. As árvores de busca nas Figuras 5(a) e 5(b) confirmam a vantagem em termos de complexidade para o algoritmo

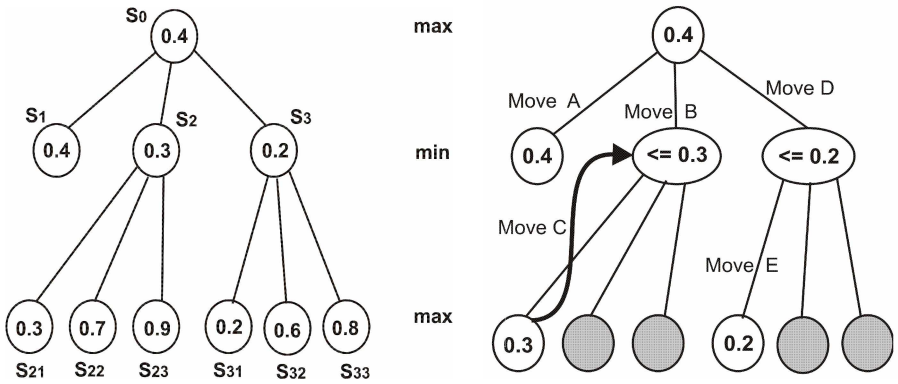


Figura 5 – Expansão da Árvore de Busca pelo Minimax e pelo Alpha-Beta fail-soft.

Alpha-Beta em relação ao Minimax. De fato, o primeiro produz o mesmo resultado, ao aconselhar o movimento A e visitar 4 nós a menos, em razão das podas.

2.4.2 Impacto da Ordenação dos Nós-Folha no Algoritmo Alpha-Beta

A possibilidade da ocorrência de podas na busca realizada pelo algoritmo Alpha-Beta faz com que o seu tempo de execução seja menor ou igual ao obtido pela busca Minimax [11]. De fato, no algoritmo Alpha-Beta, quanto mais ordenados forem os seus nós-folha mais podas ocorrerão diminuindo assim, o seu tempo de execução realizado em uma mesma árvore de busca. Em [11], os autores mostram que quando a árvore de busca está com seus nós-folha perfeitamente ordenados, o tempo de execução do algoritmo Alpha-Beta é de $O(b^{m/2})$. Por outro lado, quando a árvore de busca está com seus nós-folha ordenados aleatoriamente, o tempo de execução do algoritmo Alpha-Beta é de $O(b^{3m/4})$.

Em [17], os autores mostraram que a TT pode ser usada em associação com o AI para obter a ordenação parcial da árvore de busca. Quando o AI busca um nível mais profundo e revisita um estado, a informação da TT referente ao estado já avaliado, pode ser usado para ordenar a árvore de busca. Os dados da tabela podem ser usados desde que a profundidade do estado corrente seja compatível com a profundidade do estado recuperado na tabela (igual ou maior que a profundidade). O melhor movimento é o filho do estado (encontrado na TT) o qual tem a melhor predição durante as avaliações feitas anteriormente. A ordenação parcial dos nós consiste em fazer com que o nó de melhor movimento ocupe a primeira posição da esquerda para a direita na árvore de busca. Portanto, o melhor movimento de um estado na profundidade d, será o primeiro a ser explorado na profundidade $(d+1)$ e de acordo com Plaat [28], o melhor movimento para a profundidade d é possivelmente o melhor movimento para a profundidade $(d+1)$,

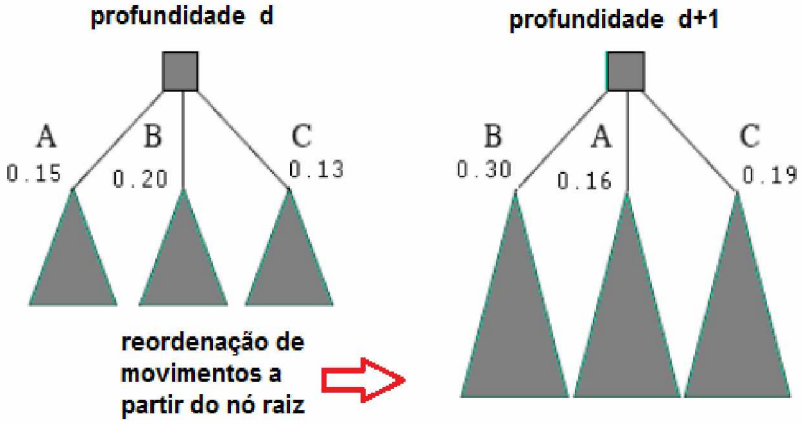


Figura 6 – Exemplo de Ordenação em uma Árvore de Busca com AI.

como mostrado na Figura 6.

Para exemplificar o impacto que a ordenação dos nós-folha causa na ocorrência das podas, são mostradas na Figura 7 (a) duas árvores (árvore-1 e árvore-2) sendo expandidas pelo algoritmo Minimax, cujos nós-folha S_1 e S_{21} estão com os valores de avaliações trocados de ordem. Nota-se que a expansão feita em ambas as árvores, explorou todos os nós e definiu como melhor escolha, o nó-filho do nó S_0 com avaliação igual a 0.4. Nesse caso, apesar dos nós escolhidos terem sido diferentes, a quantidade de nós explorados pelo Minimax não foi diferente entre as duas árvores em detrimento das ordens de seus nós-folha serem diferentes. Porém, conforme mostrado nas Figuras 7 (b) e (c), as expansões feitas pela busca Alpha-Beta em ambas as árvores foram diferentes. Na Figura 7 (b), após a exploração da ramificação mais à esquerda feita pelo Alpha-Beta, o nó S_0 da árvore-1 assume a avaliação 0.4 e o da árvore-2 assume a avaliação 0.3. Devido ao fato das avaliações dos nós-folha em ambas as árvores ser maior que o valor inicial de $Alpha = -\infty$, tal limite inferior da janela de observação no nó-raiz S_0 estreitou-se para $Alpha = 0.4$ na árvore-1 e para $Alpha = 0.3$ na árvore-2. Após a exploração do nó-folha S_{21}, na árvore-1 há uma poda $Alpha$ no nó S_{22} e no nó S_{23} mas, não há nenhuma poda na árvore-2. Isso se deve ao fato de que na árvore-1 a avaliação do nó-filho do nó minimizador S_2 é menor que o valor $Alpha$ atual e na árvore-2 não. Portanto, na árvore-2 os nós-folha S_{22} e S_{23} são expandidos mesmo que seus valores não sejam repassados para o nó S_2, evidenciando uma busca com mais nós do que se comparada com a busca feita na árvore-1. Ao finalizar a exploração de toda sub-árvore definida a partir do nó S_2, o valor de $Alpha = 0.3$ da árvore-2 é aumentado para $Alpha = 0.4$ causando um novo estreitamento na janela de observação no nó S_0. Posteriormente, a Figura 7 (c) mostra que quando a sub-árvore definida a partir do nó S_3 é explorada, em ambas as árvores ocorre as mesmas podas nos nós S_{32} e S_{33} pois, o valor de $Alpha = 0.4$ é o mesmo e a ordem de seus nós-folha a partir da sub-árvore do nó S_3 é a mesma. Finalizando toda a exploração na árvore-1 e na árvore-2, o nó escolhido como melhor opção foi o nó com avaliação 0.4. Portanto,

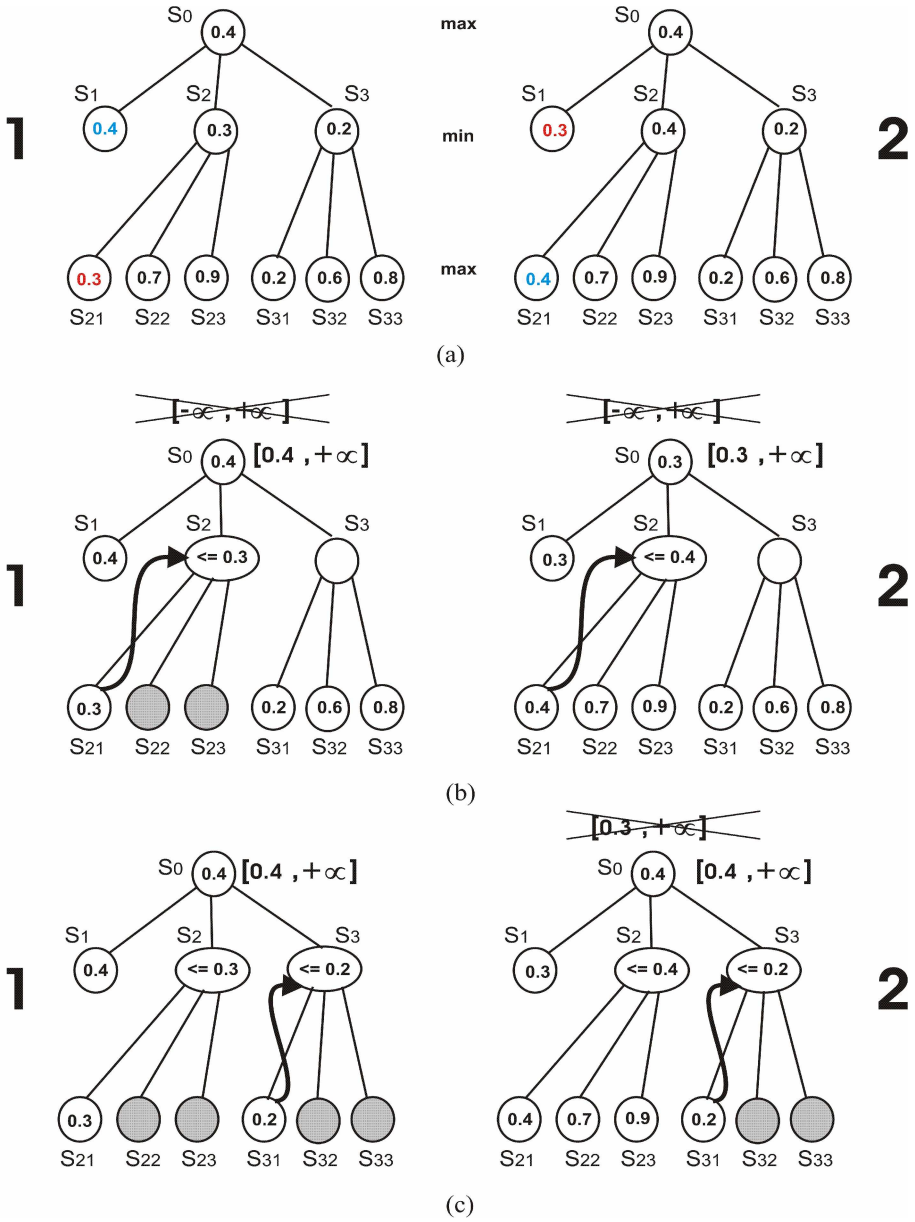


Figura 7 – Expansão Alpha-Beta em Duas Árvores com Ordenações Diferentes.

a mudança na ordem dos nós-folha fez com que o Alpha-Beta explorasse mais ou menos nós para encontrar a mesma solução do Minimax.

A dificuldade de avaliar bem o comportamento do Alpha-Beta sem que se faça um uso mais expressivo do formalismo matemático, acabou por torná-lo um ótimo objeto de estudo para os pesquisadores. Um exemplo disto pode ser visto em [29] que utilizou uma abordagem empírica para analisar o caso médio deste algoritmo. Inicialmente, os autores destacaram o fato de que as propostas realizadas até então exigiam um elevado esforço matemático para analisar o desempenho do Alpha-Beta. Assim, por meio de uma abordagem experimental associada a uma modelagem matemática probabilística, eles propuseram uma expressão matemática que calcula o número de nós-folha a serem avaliados em uma busca para o caso médio (ordenação aleatória) considerando uma certa

profundidade máxima e um certo fator de ramificação. Apesar de o trabalho ter mostrado resultados relevantes, destacou-se a ressalva de que a abordagem sugerida só é eficaz para árvores uniformes - todos os nós possuem o mesmo fator de ramificação - e cujas avaliações dos nós-folha tenham o mínimo de correlação entre si. Portanto, a proposição de uma análise empírica que permita avaliar algoritmos com comportamento mais complexo como o Alpha-Beta dando uma ênfase em simulações automáticas de modelos formais, configura uma proposta bastante interessante.

2.5 Algoritmo de Busca Distribuída PVS

Os algoritmos distribuídos foram propostos para serem processados por diferentes processadores em um mesmo intervalo de tempo com o objetivo de minimizar os valores dos tempos de execução gerados no processamento. Para obter esse ganho de desempenho, o que se faz é aumentar gradualmente a quantidade de processadores envolvidos. Em problemas que envolvem a exploração de árvores de busca, a obtenção desse comportamento nem sempre é garantida, pois, vai depender especificamente de três aspectos a saber [30] e [31]:

- sobrecarga de busca - ocorre quando o algoritmo distribuído explora nós que não seriam explorados caso a versão serial fosse usada. Por exemplo, numa versão do PVS para o algoritmo serial Minimax com podas isso pode ocorrer caso a árvore de busca não seja ordenada;
- sobrecarga de sincronização - ocorre quando um processador fica ocioso esperando algum evento ocorrer;
- sobrecarga de comunicação: ocorre da necessidade dos processadores ficarem constantemente comunicando entre si.

Para medir/avaliar o ganho de desempenho produzido nos algoritmos distribuídos em função do aumento gradual no número de processadores, há várias métricas sendo que, o *speedup* e a eficiência se destacam na preferência dos pesquisadores. O *speedup* é uma medida do grau de desempenho da distribuição. Essa medida calcula a taxa entre o tempo de execução no processamento sequencial e o tempo de execução no processamento paralelo, dada na forma:

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

, onde:

- T_s é o tempo de execução do algoritmo com apenas um processador;

- T_p é o tempo de execução do algoritmo com mais de um processador em paralelo.

A eficiência é uma medida do grau de aproveitamento dos processadores paralelos. Essa medida calcula a taxa entre o *speedup* e o número de processadores paralelos, dada na forma:

$$E = \frac{S}{p} \tag{2}$$

, onde:

- S é o *speedup* do algoritmo;
- p é o o número de processadores paralelos.

Originalmente, o algoritmo PVS [13] foi projetado para funcionar em sistemas com multi-processadores, porém, o seu uso em sistemas distribuídos com multi-computadores também foi sugerido em [16]. Quando o PVS foi criado, a ideia era paralelizar a busca serial feita pelo algoritmo Minimax [11] adaptado com podas *Alpha* e *Beta* [12]. A sua política de distribuição dos nós para os processadores é feita de forma local e estática na árvore de busca. Para isso, um processador principal é usado tanto para gerenciar a distribuição dos processadores secundários quanto fazer parte dela. A Figura 8 mostra o exemplo de uma exploração em árvore feita pelo PVS usando apenas dois processadores: o processador principal P_0 e um processador secundário P_1.

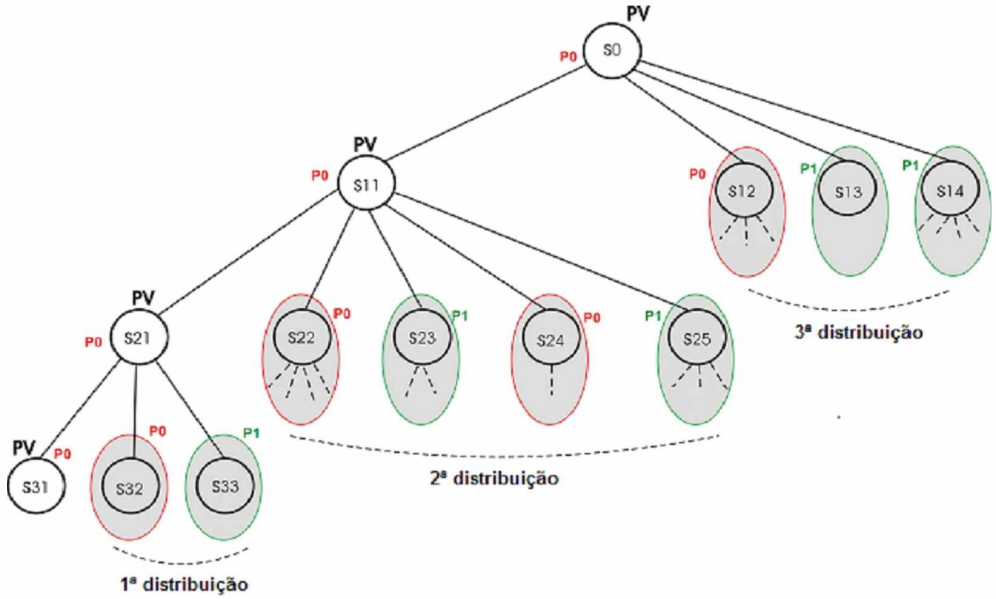


Figura 8 – Exemplo de Expansão do PVS.

De acordo com a taxonomia criada por Knuth e Moore [32], o nó-raiz e todos os nós da ramificação mais à esquerda de uma árvore são chamados de PV-nodes. Portanto, conforme mostra a Figura 8, o conjunto dos PV-nodes é dado por $\{s_0, s_{11}, s_{21}, s_{31}\}$. O

algoritmo PVS determina que a distribuição ocorre somente entre os nós-filho de um PV-node, exceto o nó-filho mais à esquerda, juntamente com suas sub-árvores. Nesse caso, as partes paralelizáveis da árvore da Figura 8 são todos os nós que estão sombreados.

Antes de ocorrer qualquer distribuição na árvore, o processador principal (P_0) deve explorar toda ramificação mais à esquerda até que se chegue no PV-node mais profundo dela (no caso, s_{31}), para depois dar início às distribuições. Na Figura 8, a primeira distribuição faz com que o processador P_0 processe o nó s_{32} e o processador P_1 processe o nó s_{33}. Concluídos esses processamentos, a segunda distribuição faz com que o processador P_0 processe o nó s_{22} e o processador P_1 processe o nó s_{23}, ambos com suas respectivas sub-árvores. A Figura 8 mostra que o nó s_{24} é processado pelo processador P_0 mas, isso só ocorre quando ele estiver ocioso. O mesmo acontece com o nó s_{25} em relação ao processador P_1. Por último, com os processadores P_0 e P_1 ociosos, ambos se tornam aptos a realizarem a terceira distribuição envolvendo os nós s_{12}, s_{13} e s_{14} com suas respectivas sub-árvores, conforme mostra a Figura 8.

Em [33], os autores analisaram algumas características importantes do comportamento do PVS a saber:

- todas as ramificações de um PV-node são exploradas, portanto, a paralelização nos nós-filho dos PV-nodes é uma boa ideia;
- para minimizar a sobrecarga de busca, a árvore explorada pelo PVS deve ser fortemente ordenada (a ramificação mais à esquerda de um nó é a melhor de todas em 70 por cento das vezes). De fato, se o Minimax estiver sujeito a podas, quanto maior a ordenação maior é a chance delas ocorrerem no processamento da ramificação mais à esquerda;
- a eficiência do PVS sempre diminui para um número de processadores maior ou igual ao fator de ramificação médio da árvore;
- a sobrecarga de sincronização é um aspecto muito forte no PVS, pois, a maioria dos processadores ociosos esperam pelo fim do processamento feito pelos processadores que lidam com ramificações "difíceis". Isso porque essas ramificações exigem grandes esforços na busca, ou seja, possuem um número bem maior de nós quando comparados com as demais ramificações.

2.6 Redes de Petri

Em sua dissertação de doutorado, Carl Adam Petri apresentou um tipo de grafo bipartido com estados associados, com o objetivo de estudar a comunicação entre autômatos [18]. Após Petri apresentar seus primeiros modelos, alguns pesquisadores, com o intuito de homenagear sua descoberta, os denominaram de Redes de Petri (RdP). As RdP

são ferramentas matemáticas, gráficas e formais que permitem a modelagem, a análise, o controle e a supervisão de sistemas dinâmicos, produzindo uma abordagem discreta e/ou contínua dos mesmos.

Os elementos básicos das RdP são: o lugar, representado por um círculo, que pode modelar um estado parcial, uma condição, uma espera, ou um procedimento; a transição, representada por uma barra ou retângulo, que modela um evento de início ou fim de um estado; o arco, representado por um arco direcionado, que liga transição(ações) a(aos) lugar(res) e vice-versa; e, por fim, a ficha ou marca, representada por um ponto em um lugar, que modela a ocorrência de um estado, conforme mostrado na Figura 9. Vale ressaltar que um lugar nunca é ligado a outro diretamente e uma transição nunca é ligada a outra diretamente, ou seja, um lugar sempre está ligado a uma ou mais transições por meio de arcos e uma transição, por sua vez, sempre está ligada a um ou mais lugares também por intermédio dos arcos [18].

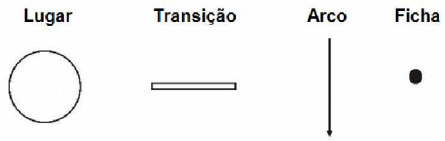


Figura 9 – Elementos Básicos de uma RdP.

As RdP são ferramentas de modelagem dinâmica, pois permitem acompanhar a evolução do sistema modelado por meio da mudança da ficha de um lugar para outro. A Figura 10 mostra um exemplo de evolução da ficha em um modelo de RdP conhecido como "Ordinário", no qual o disparo das transições que ocasionam o deslocamento da ficha de um lugar para outro é instantâneo.

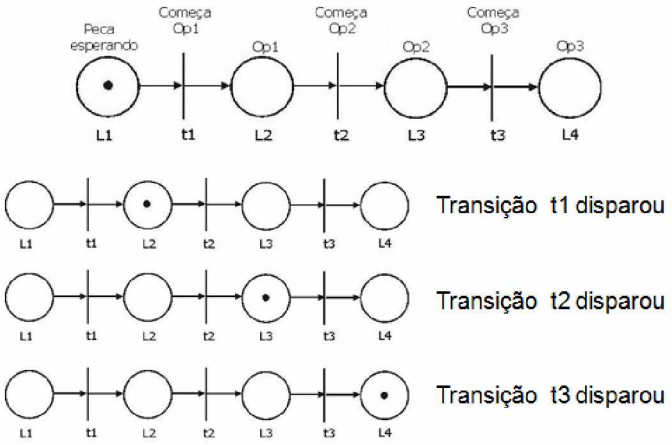


Figura 10 – Exemplo do Comportamento Dinâmico de uma RdP.

A mudança da ficha de um lugar ao outro depende do modelo utilizado que definirá uma política de disparo. Por exemplo, se as transições da RdP estão associadas a durações de tempo específicas, o disparo delas ocorre somente após o fim da duração de tempo.

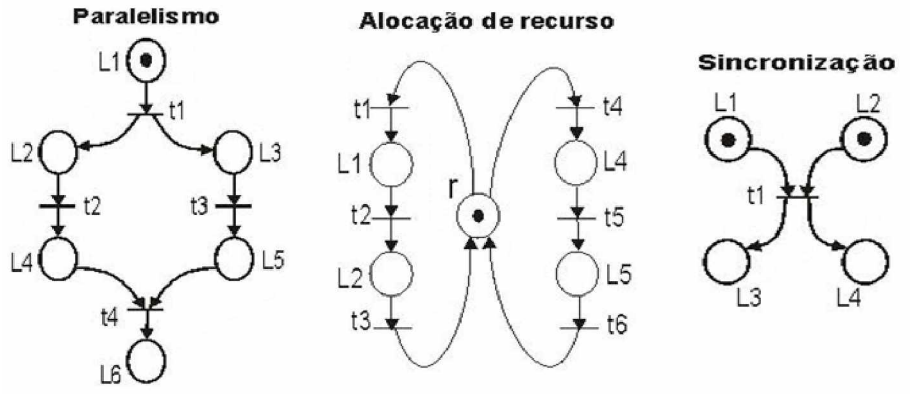


Figura 11 – Modelagem de Restrições.

Porém, se o modelo é uma RdP Predicado/Transição, o disparo das transições ocorre somente quando é satisfeita a condição definida por variáveis que estão relacionadas a estas transições. A potencialidade da modelagem por meio das RdP destaca-se em relação a outras técnicas mais tradicionais, como autômatos [7], pelo fato delas permitirem modelar restrições de alocação de recursos, sincronização e paralelismo, dentre outras, conforme mostrado na Figura 11. Sabe-se que, por meio dos autômatos, não é possível modelar de forma explícita restrições de paralelismo e nem de sincronização.

Inicialmente, as propostas de modelagem por meio das RdP concentraram-se nos problemas pertencentes à área da engenharia de um modo geral. Assim, aplicações, como sistemas de produção, manufatura, robótica, transporte, comunicação e outros com alto grau de complexidade e diversas restrições, foram cada vez mais beneficiadas com esta ferramenta. Em virtude das RdP modelarem diversas restrições, poderem gerar modelos mais compactos, utilizarem uma abordagem gráfica e formal e representarem a dinâmica do sistema modelado, houve o interesse em usá-las também em outras áreas de aplicação. Na computação, por exemplo, diversos trabalhos vêm demonstrando a eficácia no uso das RdP para resolver seus problemas. Em [34], os autores propuseram o uso de Redes de Petri Estocástica Generalizada (RdPEG), para modelar uma arquitetura paralela composta de dois processadores paralelos, contendo cada um a sua própria memória local e ambos utilizando um barramento comum para acessar dois módulos de memória compartilhada. Na área da IA, também existem algumas propostas de uso das RdP, tais como: a associação das Redes de Petri Estocástica (RdPE) com uma Rede Neural Artificial (RNA), que utiliza um aprendizado supervisionado e não-supervisionado para a estimativa de eventos raros [35] e a utilização da Lógica Nebulosa com uma RdP Distribuída para diagnosticar falhas em sistemas de potência [36].

Diversas variantes de RdP e suas extensões foram propostas ao longo dos anos e cada vez mais tais modelos tem sido usados no entendimento de organizações empresariais e em aplicações industriais. Nesse sentido, alguns especialistas passaram a dedicar seus estudos na proposição de uma norma que forneça uma referência de definição para ser usada dentro

e entre as organizações. O intuito é assegurar por meio de uma compreensão comum da técnica, que contratos empresariais referenciem a norma e que as ferramentas de suporte às RdP tenham uma interoperabilidade e sejam mais fáceis de serem criadas [37].

2.6.1 Redes de Petri de Alto Nível

No exemplo mostrado na Figura 10, a ficha da RdP não carrega nenhuma informação do sistema modelado. Isso faz com que a potencialidade da modelagem seja menos abrangente limitando seu uso em sistemas mais complexos. Porém, existem modelos de RdP que possuem esse aprimoramento de trabalhar com fichas que contêm alguma informação do problema, sendo esses modelos chamados de Redes de Petri de Alto Nível (RdPAN). Com o aumento na potencialidade da modelagem obtido por meio das RdPAN e com o consequente aumento no seu uso em diversas corporações, em 10 de maio de 2002 um grupo de pesquisadores finalizou a primeira parte do projeto de padronização internacional (ISO/IEC 15909 - Versão 4.7.3) determinando os conceitos, as definições e as notações gráficas da classe das RdPAN e de sua sub-classe Redes de Petri Predicado/Transição (RdPPT). Em 23 de junho de 2005, a segunda parte deste projeto (ISO/IEC 15909-2 WD - Versão 0.9.0) definiu um formato de transferência para as RdPAN visando a interoperabilidade das ferramentas que as usam. O formato de transferência foi baseado em *Extensible Markup Language* (XML) e chamado de *Petri Net Markup Language* (PNML). Posteriormente, em 14 de fevereiro de 2011, foi publicado o formato de transferência baseado em XML (ISO/IEC 15909-2:2011) cujo foco foram as Redes de Petri Lugar/Transição (RdPLT), as RdPAN e as Redes Simétricas (RS), além da possibilidade de inclusão de outras RdP no futuro [38]. Dentre as sub-classes de RdPAN, uma que vem sendo aplicada largamente em aplicações no meio acadêmico, industrial e empresarial são as RdPC. A extensão das RdPC conhecida como RdPCH foi escolhida para a abordagem proposta nesta tese.

2.6.2 Redes de Petri Coloridas

As RdPC são uma sub-classe das RdPAN. Segundo Aalst [39], uma RdPC é uma rede na qual cada lugar possui um tipo que define um conjunto U. Tal conjunto é um universo contendo todos os possíveis valores pertencentes a este tipo. Um multi-conjunto sobre U é então, um mapeamento $f : U \longrightarrow N$ que atribui para cada elemento $u \in U$ um número natural $f(u)$. Cada elemento $u \in U$ especifica um valor do tipo do lugar representado por U. Além disso, um arco em uma RdPC pode ter uma etiqueta (ou rótulo) associada à ele. Esta etiqueta é uma expressão com algumas variáveis que avaliam sobre um multi-conjunto. Uma transição pode ter uma *Guard* associada a ela. Esta *Guard* é uma expressão booleana que pode ter variáveis que se associam com as etiquetas dos arcos de entrada e saída da referida transição.

Uma vez que cada lugar de uma RdPC possui um tipo (cor) associado a ele, as fichas que são inseridas nesse lugar devem estar em conformidade com seu tipo que deve ser previamente declarado. Com isso, as RdPC são uma linguagem de modelagem gráfica que permite a associação da potencialidade das RdP - representação da sincronização e da concorrência em sistemas distribuídos, por exemplo - com o formalismo das linguagens funcionais, tais como, a definição e manipulação de tipos de dados [23].

2.6.3 Definição Formal

Uma RdPC é definida formalmente com sendo uma tupla $CPN = (\Sigma; P; T; A; N; C; G; E; I)$ [40], onde:

- Σ é um conjunto finito de tipos não-vazios, chamado de conjunto de cores;
- P é um conjunto finito de lugares;
- T é um conjunto finito de transições;
- A é um conjunto finito de arcos tal que:
  - $P \cap T = P \cap A = T \cap A = \oslash$.
- N é uma função nó. Ela é definida de A em:
  - $PXT \cup TXP$.
- C é uma função cor. Ela é definida de P em Σ;
- G é uma função *Guard*. Ela é definida de T em expressões tal que:
  - $\forall t \in T : [Type(G(t)) = Bool \wedge Type(Var(G(t))) \subseteq \Sigma]$.
- E é uma função expressão de arcos. Ela é definida de A em expressões tal que:
  - $\forall a \in A : [Type(E(a)) = C(p(a))_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma]$, onde $p(a)$ é um lugar de $N(a)$.
- I é uma função inicialização. Ela é derivada de P em expressões fechadas tal que:
  - $\forall p \in P : [Type(I(p)) = C(p)_{MS}]$.

A existência de uma definição formal para as RdPC é muito importante porque:

1. é a base para a simulação, ou seja, para a execução delas;
2. é a base para métodos de verificação formal (espaços de estados e invariantes de lugar);

3. é indispensável para produzir uma classe de rede bem formada;

Apesar da importância dessa formalização, para um usuário ela não é tão relevante por dois motivos [40]: o primeiro é que a sintaxe correta deve ser checada pelo editor de RdPC (ferramenta de computador na qual a RdPC é construída); e o segundo é que o uso correto da semântica deve ser garantida pelo simulador de RdPC usado. A seção 2.6.6 apresentará as principais características do ambiente gráfico CPN Tools usado como ferramenta de edição e de simulação para os modelos gerados nesta tese.

2.6.4 Exemplificando a Dinâmica de um Modelo de RdPC

Uma grande característica das RdPC em relação a outros modelos, é a capacidade de redução estrutural da modelagem produzida. Para mostrar esse diferencial, considera-se um exemplo de um sistema de produção de peças que produz, por hora, 3 peças $pc1$, 2 peças $pc2$ e 4 peças $pc3$. A Figura 12 mostra como uma RdP Ordinária modela esse sistema. Os lugares $E1$ e $S1$ modelam, respectivamente, os estados anterior e posterior à produção da peça $pc1$. As fichas contidas no lugar $E1$ definem as peças $pc1$ que serão produzidas em uma hora. Cada disparo da transição $Inicio1$ define o início da produção de uma peça $pc1$ e cada disparo da transição $Fim1$ define o fim da produção de uma peça desse tipo. O lugar $PC1$ modela o estado de produção de uma peça do tipo $pc1$. O mesmo raciocínio vale para as outras partes do modelo em relação às peças $pc2$ e $pc3$.

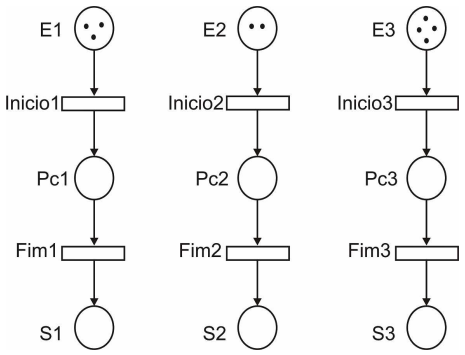


Figura 12 – Modelo de RdP Ordinária para o Sistema de Produção de 3 Peças.

Supondo que o sistema de produção apresentado seja tal que, há 3 linhas de produção distintas (uma para cada tipo de peça) e que na mesma linha de produção, só pode ser produzida uma peça por vez. Essa característica não pode ser representada no modelo da Figura 12, pois não há nenhum controle sobre a política de disparo das transições de início e fim de produção das peças. Para isto, o modelo da Figura 13 apresenta a política de controle exigida com a inclusão dos lugares $K1$, $K2$ e $K3$. Observa-se que o disparo da transição $Inicio1$ está condicionado ao fato de que deve haver ficha tanto no lugar $E1$ quanto em $K1$.

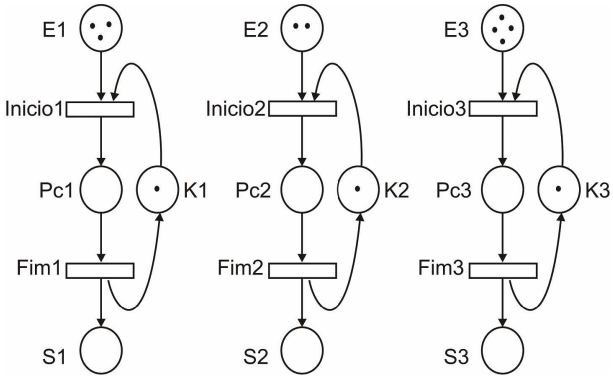


Figura 13 – Modelo de RdP Ordinária com Política de Disparo.

Para entender o efeito da representação dessa política de disparo no modelo, na Figura 14 (a) é mostrado o estado imediatamente anterior ao disparo da transição $Inicio1$ que, ao ser disparada (Figura 14 (b)), faz com que uma ficha seja retirada dos lugares $E1$ e $K1$ e produzida no lugar $PC1$ (produção de uma peça $pc1$). Nota-se que, a transição $Inicio1$ não pode disparar novamente porque o lugar $K1$ não possui ficha. Após o disparo da transição $Fim1$, uma ficha é produzida nos lugares $S1$ e $K1$ (Figura 14 (c)). A partir desse momento, o modelo mostra que a transição $Inicio1$ pode ser disparada novamente indicando que uma nova peça $pc1$ será produzida.

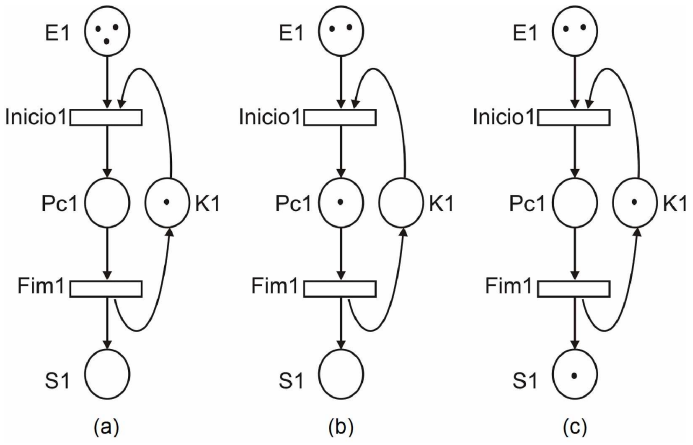


Figura 14 – Evolução Dinâmica do Modelo de RdP Ordinária com Política de Disparo.

É importante observar que no modelo da Figura 13, para três tipos de peças, foram necessários 12 lugares, 6 transições e 18 arcos. Sabendo que muitos sistemas reais de produção fabricam dezenas de produtos e usando o mesmo raciocínio, se fosse considerado um sistema de produção de dez tipos distintos de peças seriam necessários 40 lugares, 20 transições e 60 arcos. É evidente o aumento da complexidade da estrutura do modelo com o uso de uma RdP Ordinária. O que ocorre é que as fichas usadas não apresentam nenhuma informação do sistema com relação aos tipos de peças produzidas, ficando para a estrutura do modelo o encargo de distingui-las. Porém, nas RdPC esse problema é amenizado uma vez que as fichas que fluem no modelo possuem atributos (cores) e ins-

tâncias (valores) que permitem reduzir a estrutura do modelo sem que haja perda na representação do sistema.

A Figura 15 (a) mostra a redução obtida com o uso de uma RdPC para modelar o mesmo sistema de produção de três peças, onde foram necessários apenas 4 lugares, 2 transições e 6 arcos. A Figura 15 (a) representa o estado imediatamente anterior ao primeiro disparo da transição *Inicio*. É importante destacar que em uma RdPC um lugar pode conter n fichas repetidas (multiplicidade de valor), de modo que, $n \geq 0$. A esse conjunto de fichas com multiplicidade de valor denomina-se **multi-conjunto**. Na Figura 15 (a), a marcação inicial do modelo indica que o lugar E possui 9 fichas, sendo: 3 fichas com o valor igual a 1 (indicando que 3 peças $pc1$ serão produzidas), 2 fichas com o valor igual a 2 (indicando que 2 peças $pc2$ serão produzidas) e 4 fichas com o valor igual a 3 (indicando que 4 peças $pc3$ serão produzidas). Já o lugar K possui 3 fichas, sendo: 1 ficha com o valor igual a 1 (indicando a linha de produção para produzir peças $pc1$), 1 ficha com o valor igual a 2 (indicando a linha de produção para produzir peças $pc2$) e 1 ficha com o valor igual a 3 (indicando a linha de produção para produzir peças $pc3$).

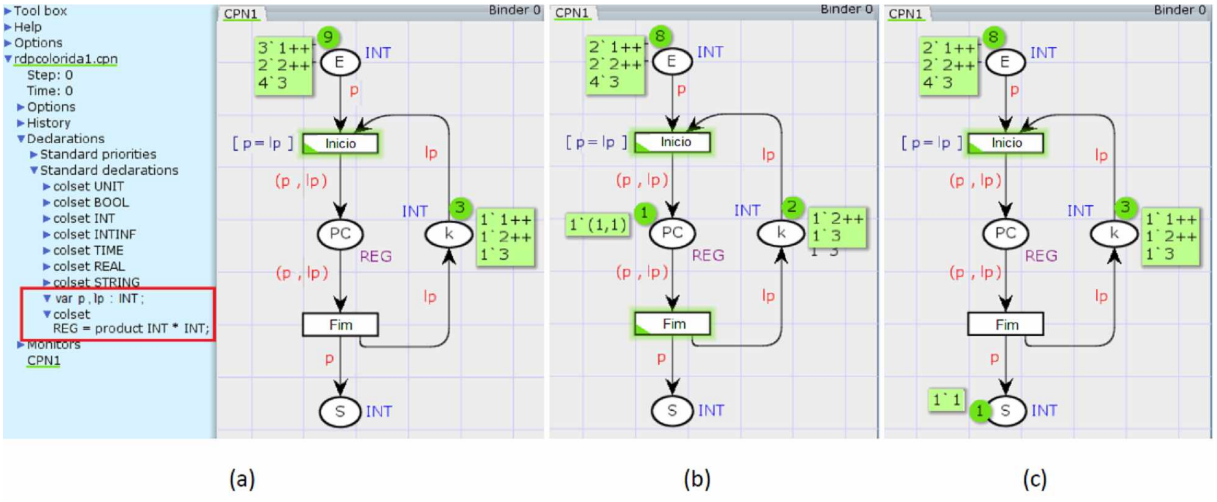


Figura 15 – Modelo de RdPC para um Sistema de Produção de 3 Peças.

Todos os arcos da RdPC da Figura 15 (a) possuem inscrições chamadas de "Etiquetas" (ou rótulos). Tais rótulos podem ser variáveis ou funções, ambas capazes de carregar as instâncias das fichas de um lugar a outro. Além disso, as funções podem também alterar essas instâncias durante o fluxo das fichas no modelo. Para que as variáveis e as funções funcionem corretamente é necessário que elas sejam declaradas com tipos de dados apropriados. Por exemplo, conforme mostrado na Figura 15 (a), os lugares E e K possuem a cor INT associada a eles, sendo suas fichas instanciadas com valores inteiros. Quando a transição *Inicio* disparar, uma ficha será removida de cada uma desses lugares e seus valores devem ser "carregados" durante essa dinâmica. Portanto, os arcos de saída do lugar E e do lugar K são associados à variável p e lp, respectivamente, cujas declarações

são também do tipo INT. Observa-se que a transição *Inicio* possui uma condição lógica $[p = lp]$ associada a ela, chamada de *Guard*. Caso a *Guard* seja verificada como verdadeira, ela produz um resultado *true*, provocando o disparo da transição. Caso contrário, a *Guard* produz um resultado *false*, não permitindo o disparo da transição. No exemplo da Figura 15 (a), a transição *Inicio* está apta ao disparo porque a *Guard* $[p = lp]$ é verdadeira, ou seja, há uma ficha no lugar E com o mesmo valor que outra ficha no lugar K. O uso da *Guard* garante que a fichas removidas dos lugares E e K após um disparo da transição *Inicio* tenham seus valores iguais. Isso modela a característica do sistema de produção de peças que diz que cada linha de produção é responsável por produzir apenas um tipo de peça, ou seja, se a variável p é igual a 1 (peça $pc1$) então, a variável lp deve ser igual a 1 (linha de produção da peça $pc1$) para que a transição *Inicio* seja disparada.

Outro ponto importante a ser observado é que o arco de saída da transição *Inicio* está associado à etiqueta (p, lp). Tal etiqueta realiza a composição das variáveis p e lp de modo que a ficha a ser produzida no lugar PC conterá a composição dos valores das duas variáveis em um único tipo. Esse tipo foi declarado como $REG = productINT * INT$, conforme mostrado no lado esquerdo da Figura 15 (a). De acordo com a Figura 15 (b), com o disparo da transição *Inicio* uma ficha com o valor $(1, 1)$ foi produzida no lugar PC. Nota-se que o lugar PC possui a cor REG associada a ele, permitindo-o receber fichas com instâncias compostas.

Vale ressaltar que nenhuma ordem de prioridade de disparo das transições foi estabelecida no modelo. De fato, na Figura 15 (b) tanto a transição *Inicio* quanto a transição *Fim* estão aptas ao disparo, cabendo ao ambiente CPN Tools decidir qual delas será a próxima transição a ser disparada. As RdPC podem ser aprimoradas com alguns recursos adicionais que estabelecem a ideia de prioridade de disparo das transições, tais como: arcos inibidores, associação de tempo e valores de prioridades [23, 39]. A Figura 15 (c) mostra que a transição disparada foi a transição *Fim*, representando o fim da produção de uma peça $pc1$. Nota-se que os arcos de entrada dos lugares S (contendo a cor INT) e K (contendo a cor INT) desfazem a composição feita anteriormente uma vez que suas etiquetas, respectivamente, possuem as variáveis p e lp separadas.

2.6.5 Redes de Petri Coloridas Hierárquicas

Além da compactação do modelo obtida com o uso de uma RdPC, é possível tornar a representação desse modelo ainda mais "amigável" ao usuário por meio da técnica de hierarquização. A ideia é a mesma usada na evolução das linguagens de programação, quando se propõe o uso de rotinas para modularizar o código original. No caso da modelagem, é possível estender uma RdPC, ao adicionar-lhe a ideia de hierarquias, culminando, assim, nas RdPCH [23]. O que se faz é decompor o modelo global da RdPC em sub-redes secundárias, denominadas de "páginas". Portanto, uma rede principal estará em um nível hierárquico mais abstrato, caracterizando a estrutura mais geral, e esta poderá

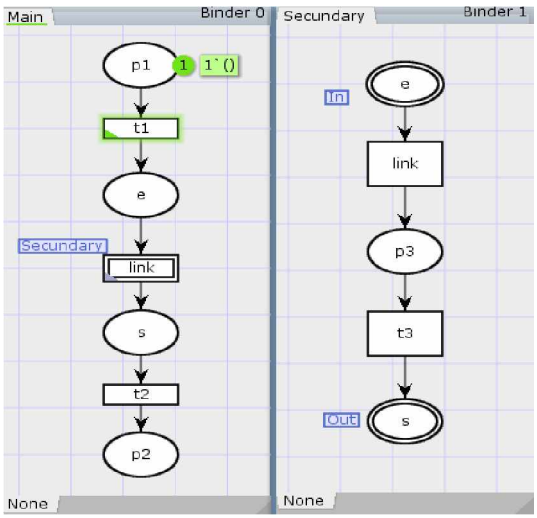
se comunicar com as outras sub-redes situadas nas páginas (níveis hierárquicos menos abstratos).

A comunicação do nível mais abstrato do modelo com as páginas só é possível por meio de uma "transição abstrata" (ou transição de substituição) representada por bordas duplas. Quando uma transição abstrata é criada em uma rede principal, ela é automaticamente clonada em uma sub-rede (página), formando um *link* entre elas.

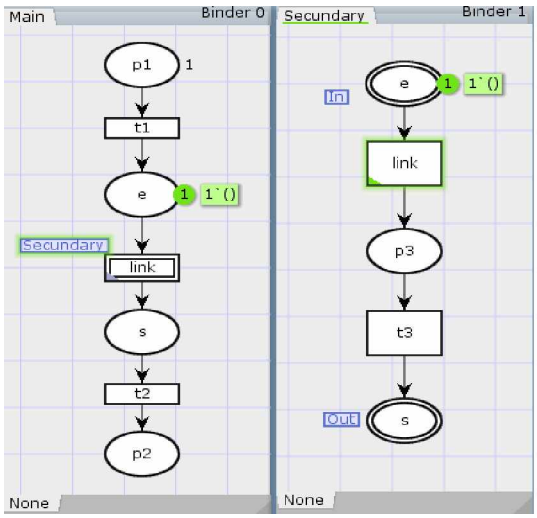
Como funciona a evolução dinâmica entre uma rede principal e uma sub-rede? Considerando o modelo da Figura 16(a), no qual a marcação inicial mostra uma ficha no lugar $p1$ sem cor pré-definida, observa-se que há uma rede principal chamada *Main* e uma sub-rede chamada *Secundary*. A transição abstrata *link* estabelece o elo entre elas. Nota-se que, na sub-rede *Secundary*, há um lugar com borda dupla (e) associado a um *socket* de entrada (*In*) e há outro lugar com borda dupla (s) associado a um *socket* de saída (*Out*). Conforme mostrado na Figura 16(b), quando uma ficha é produzida no lugar e da rede *Main*, ela também é produzida no lugar e da sub-rede *Secundary* função da transição abstrata *link*. Em seguida, com o disparo da transição *link* da sub-rede *Secundary*, é produzida uma ficha no lugar $p3$ (Figura 16(c)). Ao disparar a transição $t3$, a ficha que é produzida no lugar s da sub-rede *Secundary*, também é produzida no lugar s da rede *Main*, refletindo o elo formado entre a rede principal e a sub-rede (Figura 16(d)). Por fim, a transição $t2$ da rede *Main* é disparada e a ficha finalmente é produzida no lugar $p2$ (Figura 16(e)). Vale destacar que, com a criação da transição abstrata *link*, os *sockets* *In* e *Out* foram inseridos na sub-rede *Secundary* para permitirem as fichas de entrarem e saírem dessa sub-rede.

Além de a RdPCH permitir que uma rede principal se comunique com uma sub-rede, é possível uma sub-rede se comunicar com outra sub-rede (em qualquer nível hierárquico). Isso é similar à comunicação existente entre uma rotina e outra em uma linguagem de programação. No modelo, o recurso que permite tal efeito é chamado de "lugar de fusão". O que se faz é associar uma fusão (link) de um lugar situado em uma sub-rede a outro lugar situado em outra sub-rede. A fusão faz com que ambos os lugares se comportem como clones um do outro, ou seja, se uma ficha é produzida em um dos lugares, imediatamente ela é produzida no outro lugar com a mesma fusão. O mesmo vale para a remoção dessa ficha em tais lugares.

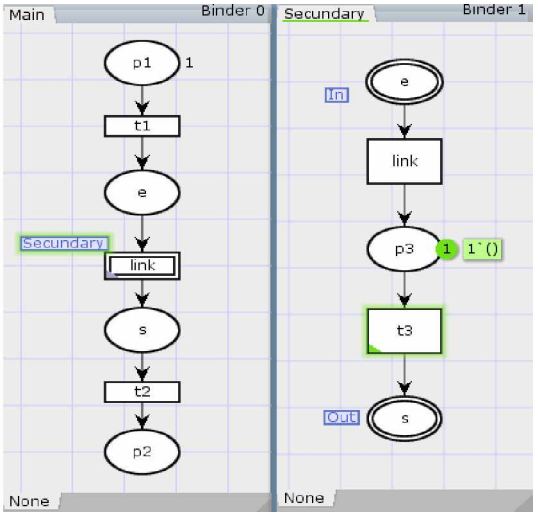
A Figura 17 (a) mostra um exemplo no qual duas sub-redes (*Secundary*1 e *Secundary*2) comunicam-se por meio de lugares de fusão. Quando um ficha é produzida no lugar de fusão $p3$ (*Fusion1*) da sub-rede *Secundary*1 (Figura 17 (b)), uma cópia dela também é produzida no lugar de fusão $p3$ (*Fusion1*) da sub-rede *Secundary*2. Isso reflete o fato de que o fluxo da ficha passou da sub-rede *Secundary*1 para a sub-rede *Secundary*2. É importante observar que tal mudança fez com que a transição $t3$ da sub-rede *Secundary*2 se tornasse apta ao disparo (Figura 17 (c)). Quando a transição $t3$ for disparada, a ficha será produzida no lugar de fusão $p4$ (*Fusion2*) de ambas as sub-redes (*Secundary*1 e



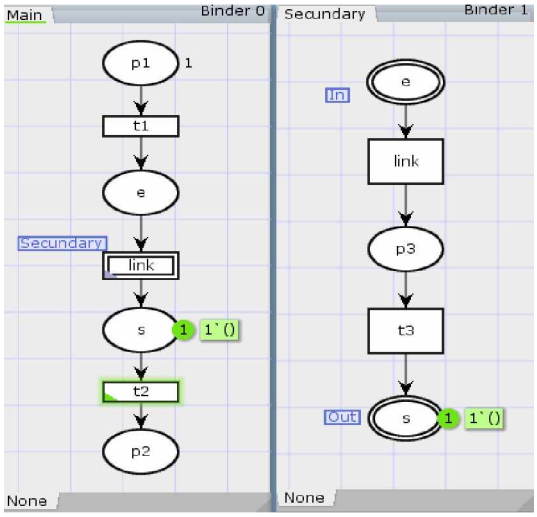
(a)



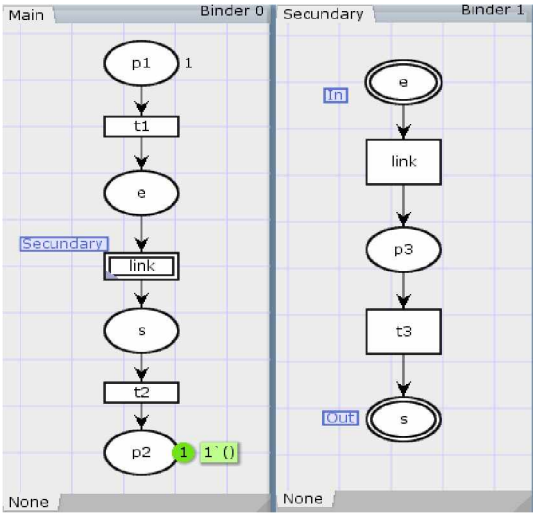
(b)



(c)



(d)



(e)

Figura 16 – Exemplo da Evolução Dinâmica entre uma Rede e uma Sub-Rede

*Secundary*2), indicando que o fluxo da ficha voltará para a sub-rede *Secundary*1.

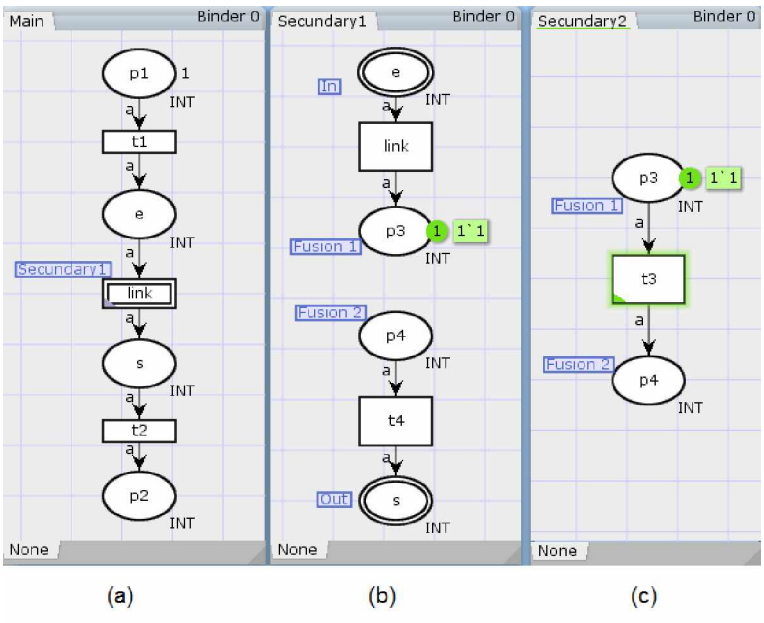


Figura 17 – Exemplo de Comunicação entre Duas Sub-Redes.

2.6.6 CPN Tools

Um ambiente de modelagem e simulação que tem contribuído muito com a "popularização" do uso das RdPC e de suas extensões é o CPN Tools [24]. Esse ambiente é composto por um conjunto de ferramentas que permitem ações, tais como editar, simular, analisar espaços de estados e avaliar o desempenho dos modelos criados. A Figura 18 mostra a versão 4.0.0 do CPN Tools onde o usuário pode trabalhar explorando um ambiente gráfico com diversos recursos. Além disso, é possível o uso combinado da programação funcional com o modelo a ser criado de forma bem intuitiva e prática. Outro diferencial dessa ferramente é que ela é distribuição na forma *free*.

Detalhando melhor a Figura 18, observa-se que o CPN Tools possui uma aba principal de edição que se apresenta de forma quadriculada permitindo a unificação dos elementos que irão compor o modelo de maneira mais organizada. Além disso, o ambiente disponibiliza as seguintes abas: *Aux, Create, Declare, Hier, Net, View* e *Style* que auxiliam na construção dos modelos; *Sim* que permite a execução passo a passo ou ininterrupta dos modelos criados; e *Mon* e *SS* que permitem o acompanhamento detalhado e análise dos estados obtidos com as simulações dos cenários. Em [41], mostrou-se através de exemplos práticos, as vantagens obtidas pelo usuário do CPN Tools no processo de modelagem e análise de desempenho dos modelos. Em especial, os autores citaram exemplos de como obter uma análise do espaço de estados obtido com a execução do modelo, bem como, os seus respectivos relatórios contendo dados estatísticos.

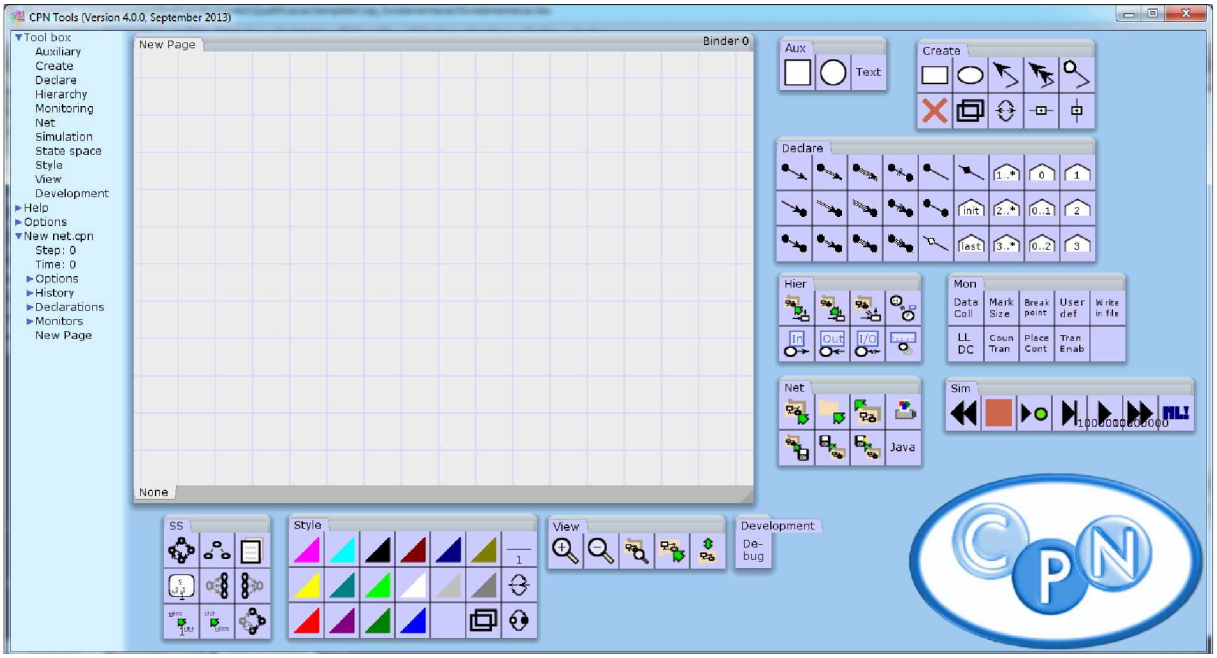


Figura 18 – Versao 4.0.0 do CPN Tools.

2.6.7 Linguagem *Standard* ML

O CPN Tools usa a linguagem CPN ML [42] para especificar declarações e inscrições nos modelos de RdPC. Esta linguagem é uma extensão da linguagem de programação funcional *Standard* ML, desenvolvida na universidade de Edinburgo. Essa extensão é definida de três formas:

- A primeira extensão "suaviza" a sintaxe das declarações do conjunto de cores (tipos);
- A segunda extensão permite ao usuário declarar suas próprias variáveis usando tipos básicos ou estruturados;
- A terceira extensão permite ao usuário declarar variáveis de referência.

Os tipos básicos da linguagem *Standard* ML são:

- **int** que representa valores inteiros;
- **real** que representa valores reais;
- **string** que representa *strings*;
- **bool** que representa valores booleanos;
- **unit** que tem apenas o valor sem tipo associado.

A Tabela 1 mostra os operadores básicos da linguagem *Standard* ML. Vale destacar que as expressões formadas pela combinação de operadores podem fazer uso de parênteses para indicar uma prioridade de um operador em relação à outro. Outro ponto importante a ser observado é que *strings* são escritas entre aspas duplas.

Tabela 1 – OPERADORES BÁSICOS DA LINGUAGEM STANDARD ML

Operadores Básicos	**Descrição**
	Operador unário negativo para **int** e **real**
+, -	Operadores de adição e subtração para **int** e **real**
*	Operador de multiplicação para **int** e **real**
div, mod	Operadores de divisão e módulo para **int**
^	Operador de concatenação para *string*
=,<>, <, >, <=, >=	Operadores de comparações básicas aplicáveis para a maioria dos tipos
not	A negação lógica
andalso	AND lógico
orelse	OR lógico
if then else	Operador ternário condição

Uma vez que os lugares de uma RdPC são "tipados", isto significa que todas as fichas em um lugar tem um valor de um tipo comum. Consequentemente, todos as fichas pertencentes a um dado lugar devem pertencer ao mesmo conjunto de cor (tipo). Na linguagem CPN ML há o conjunto de cores simples e o conjunto de cores compostas, sendo que o primeiro é definido pelos tipos básicos da linguagem *Standard* ML. Portanto, o conjunto simples de cores é definido da seguinte forma:

- colset INT = int;
- colset STR = string;
- colset BOOL = bool;
- colset UNIT = unit;

Não há um conjunto de cor para reais, pois, no CPN Tools a igualdade não está definida para os números reais. Portanto, em funções e expressões, os reais não podem ser usados a menos que sejam convertidos em inteiros (por meio de arredondamento) ou strings.

Usando os tipos básicos, é possível construir subconjuntos dos conjuntos simples de cores. Assim, ao declarar:

- *colset Age = int with 0..130;* (somente os valores 0,1,2,...,130 são permitidos);
- *colset Temp = int with -30..40;* (somente valores inteiros entre -30 e 40 são permitidos);
- *colset Alphabet = string with "a"."z";* (somente letras minúsculas do alfabeto são permitidas);

Além dos conjuntos simples de cores, a linguagem *CPN* ML também suporta a definição de novos e mais complexos conjuntos de cores obtidas a partir das cores simples. Tais conjuntos compostos de cores podem ser definidos usando os construtores de tipo: *product*, *record* e *list*.

No ambiente CPN Tools, com a declaração dos conjuntos de cores, o usuário normalmente faz é associar ao modelo de RdPC funções que manipulem informações associadas a esses conjuntos de modo a produzirem os resultados esperados. As funções podem receber e/ou enviar valores por meio de variáveis e além do uso de operadores, elas podem manipular funções nativas da linguagem *Standard* ML e/ou funções criadas pelo próprio usuário. Maiores detalhes das declarações e manipulações de conjuntos de cores e funções podem ser vistos em [39].

Capítulo 3

Trabalhos Correlatos

Diversos trabalhos propuseram o uso de alguma técnica de análise de desempenho de algoritmos. Porém, muitos deles propõem o uso de técnicas puramente matemáticas que, dependendo do algoritmo, configuram um esforço bastante árduo ou até mesmo ineficaz em função do comportamento atípico desse algoritmo. Para auxiliar na organização da apresentação dos trabalhos correlatos, este capítulo está divido em três seções. Cada uma delas relata alguns trabalhos correlatos que propuseram a análise de desempenho:

1. de softwares baseada em métodos empíricos;
2. de softwares baseada em ferramentas de modelagem visual;
3. de algoritmos baseada em redes de Petri;

3.1 Análise de Desempenho de Softwares Baseada em Métodos Empíricos

Para amenizar o enfoque matemático nas análises, algumas propostas sugeriam o uso de métodos empíricos para a realização da análise de programas implementados. Em [3] os autores propuseram um método empírico baseado na complexidade computacional para entender a escalabilidade de programas. Para este caso em particular, entende-se por escalabilidade a capacidade de ganho de desempenho do programa em função do incremento de novas cargas de trabalho [4]. Assim, a abordagem feita em [3] executa um determinado programa considerando diversas cargas de modo a obter e usar informações provenientes do programa em um modelo. Tal modelo prevê o desempenho do programa em função do tamanho da carga de trabalho. De posse do modelo, o intuito é comparar os resultados das simulações com as expectativas do programador ou com os limites teóricos assintóticos de modo a revelar *bugs* de desempenho ou confirmar que o desempenho do programa é escalonado conforme o desejado. A ferramenta usada para gerar e simular o modelo (*Trend-prof*) é de propriedade dos autores.

Outra proposta interessante foi sugerida em [5]. O método empírico proposto é baseado na comparação de desempenho de modelos de programação paralelos para avaliar o *speedup*, a eficiência e a escalabilidade usando algoritmos Minimax de busca paralela em árvores de jogos geradas no jogo *tic-tac-toe* (jogo da velha). O modelo computacional paralelo proposto pelos autores explora o particionamento da árvore em largura para cada nível da árvore de jogo e é baseado na combinação dos paradigmas algorítmicos paralelos "gestor-trabalhador" e "iterações assíncronas". A comparação de desempenho foi feita para três modelos de programação paralelo [4]: *multithreaded*, *flat* (baseado na interface *Message Passing Interface* (MPI)) e híbrido (MPI+OpenMP). O *speedup* e a eficiência, bem como, a escalabilidade (em relação ao incremento ou alteração de recursos no computador) e seu impacto no desempenho do sistema paralelo foram estimados com base em resultados experimentais e comparados por meio da análise de gráficos.

Diferentemente dos trabalhos citados nesta seção, a abordagem sugerida nesta tese não exige que os algoritmo estejam implementados em alguma linguagem de programação.

3.2 Análise de Desempenho de Softwares Baseada em Ferramentas de Modelagem Visual

Além dos métodos empíricos citados na seção 3.1, alguns pesquisadores sugeriram o uso de ferramentas de modelagem visual também com o intuito de propor alternativas para o aprimoramento de sistemas computacionais sem focar em métodos matemáticos. Em [6] a proposta se concentrou em usar grafos no desenvolvimento de protocolos de roteamento em redes dinâmicas de modo a propiciar menores rotas e maiores desempenhos no tráfico da rede.

Na abordagem sugerida por [8] os autores introduziram um método de modelagem de comportamento de software baseado em autômatos de estados finitos estendidos e em métodos de modelagem de comportamento de software existentes e focado em restrições sobre valores de dados e interações entre componentes de software. A modelagem de comportamento de software pode ser utilizada para detectar comportamentos anômalos, verificar protocolos, gerar casos de teste, capturar sequências de eventos inesperados e verificar a compatibilidade das propriedades do programa. Os autores combinaram as ferramentas *Daikon* e *ESC/JAVA* para obter as restrições e apresentá-las em arestas de autômatos de estados finitos estendidos para que a precisão do modelo construído pudesse ser melhorada. Os resultados das experiências mostraram que este modelo pode capturar muitas informações mais precisas e fornecer uma boa garantia para a análise, verificação e teste de software.

Já em [10] sugeriu-se o uso da UML para modelar e visualizar as interações dinâmicas entre objetos em uma aplicação em C++ de modo a possibilitar que o usuário altere ou exclua partes do programa tornando-o mais "sintético" sem comprometer a aplicação cujas

propriedades estáticas e dinâmicas foram exploradas por meio dos diagramas UML. Em especifico, os diagramas de sequência e comunicação foram responsáveis na visualização das interações dos objetos.

As abordagens apresentadas possuem limitações quando o alvo é a análise de desempenho de algoritmos: os grafos e os autômatos possuem um enfoque predominantemente estático na sua forma de representação que compromete o processo de avaliação do tempo de execução por meio de simulações automáticas; já a UML, apesar de contar com alguns recursos técnicos para lidar com modelos dinâmicos - os diagramas de atividades e de sequência [9] - não provê recursos que permitam, por exemplo, representar a dinâmica completa de algoritmos distribuídos.

3.3 Análise de Desempenho de Algoritmos Baseada em Redes de Petri

Alguns trabalhos propuseram o uso das RdP para modelar tipos específicos de comandos usados em algoritmos. Em [21], o foco foi exclusivamente modelar comandos de atribuição e iterativos por meio de modelos de RdP Ordinárias. A semântica de modelagem sugerida propõe que cada comando do algoritmo seja modelado por uma transição. Além disso, o autor propôs que, nos modelos, cada estrutura de dados é representada por uma ficha situada em um lugar que corresponde a um estado específico de processamento. O modelo de RdP escolhido (grafo de eventos) inviabilizou a representação de comandos condicionais do tipo IF-THEN-ELSE.

Um número limitado de ideias tem emergido para modelar algoritmos recursivos com RdP. Em [20], a chamada e o retorno recursivos foram representados por meio de um modelo de RdP Ordinário. Porém, uma limitação desta abordagem é o fato de que a semântica associada às fichas é bastante restrita não permitindo representar algoritmos com estruturas mais complexas, como por exemplo, algoritmos com diversos desvios nos fluxos de controle e/ou algoritmos distribuídos. Em [19], Kissoum usou uma RdPC recursiva gerada a partir de um conjunto de regras de transição aplicadas a um diagrama de atividades. Neste caso, o autor mostrou que o modelo de RdP escolhido permitiu a representação de processos repetitivos como sendo processos recursivos. Porém, essa proposta tem o inconveniente de requerer a priori a construção de um diagrama UML (modelo estático) antes de gerar o modelo de RdP desejado (modelo dinâmico).

No contexto dos algoritmos distribuídos, uma proposta de uso das RdP foi sugerida em [22]. Neste trabalho, os autores usaram uma RdPC juntamente com o ambiente CPN Tools para a detecção e correção de erros que eventualmente existem em uma implementação de algoritmo distribuído. Para isto, foi proposto o estudo de caso de algoritmos distribuídos cujas implementações (contendo erros de *deadlock*) fazem uso da interface MPI como ambiente de troca de mensagens. Inicialmente os autores usaram o CPN Tools

para criar o modelo que representasse apenas a comunicação de dois processos paralelos do algoritmo distribuído. Na sequência, este modelo foi usado para detectar a ocorrência dos erros de *deadlock*.

Capítulo 4

Avaliação da Função de Complexidade dos Algoritmos por meio das RdPCH

Neste capítulo, é explicitada a primeira etapa de desenvolvimento desta tese que consiste na avaliação da função de complexidade dos algoritmos por meio de simulações automáticas de modelos de RdPCH, usando o ambiente CPN Tools. A principal motivação para o uso dessa técnica é a validação da modelagem das estruturas e dos fluxos de controles de diversos comandos existentes nos algoritmos, especialmente, aqueles que provocam desvios intensos nos fluxos de dados, tais como os comandos condicionais, iterativos e recursivos. Para tornar possível as simulações automáticas do modelo no ambiente CPN Tools, foram implementadas no modelo algumas funções de controle apropriadas e foram usados operadores de tempo com custo unitário associados às transições que modelam os comandos do algoritmo. Com o modelo finalizado, simulou-se automaticamente diversos cenários distintos comparando a curva obtida com a função de complexidade teórica do algoritmo de modo a confirmar a validação da abordagem desta etapa.

4.1 Modelagem dos Comandos

A semântica de modelagem utilizada é a mesma sugerida por Hanzálek em [21], em que cada comando do algoritmo é modelado por uma transição. Além disso, o autor propôs que, nos modelos, cada estrutura de dados é representada por uma ficha situada em um lugar que corresponde a um estado específico de processamento. Apesar da semântica ser a mesma, a abordagem de Hanzálek não é apropriada para modelar comandos condicionais e recursivos pelo fato do autor ter usado um modelo mais simples de RdP do que o modelo de RdPCH usado nesta tese.

4.1.1 Comando de Atribuição

A Figura 19 (a) mostra uma sequência de dois comandos de atribuição, cujas modelagens estão representadas pelas transições $X := 2$ e $Y := X + Z$, respectivamente, conforme mostrado na Figura 19 (b). Ao ocorrer o disparo sucessivo dessas transições, a ficha situada inicialmente no lugar p_1 (Figura 19 (b)) é deslocada para os lugares p_2 (Figura 19 (c)) e p_3 (Figura 19 (d)). Esse fato simula as execuções sequenciais dos comandos de atribuição modelados. A possibilidade de acompanhar, no modelo, a dinâmica de execução dos comandos, é um ganho que se obtém com o uso das RdP.

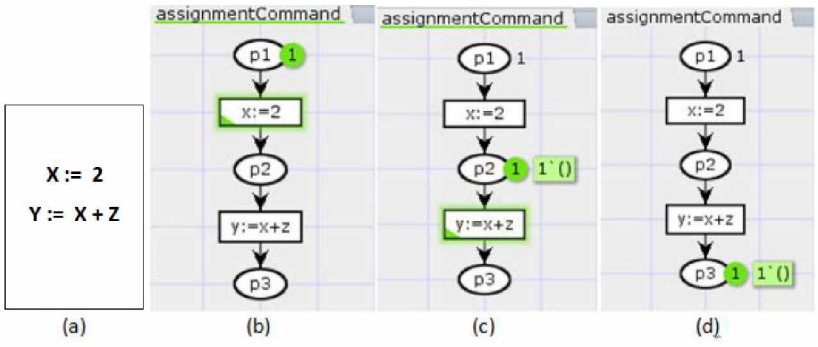


Figura 19 – Modelagem de Comandos de Atribuição.

4.1.2 Comandos Condicional e Iterativo

A modelagem e a simulação automática de comandos em que a execução real envolve desvios no fluxo de controle, como os comandos condicional, iterativo e recursivo, requerem habilidades específicas da ferramenta de modelagem usada. As Figuras 20 (b), (c) e (d) representam exemplos de modelagem de comandos condicional e iterativo. O fluxo de controle do comando condicional mostrado na Figura 20 (a) (linhas 1 - 4) é ilustrado nas Figuras 20 (b) e (c), sendo que a primeira representa a situação em que a condição $(x > 0)$ do comando é verdadeira (disparando a transição t_1) e, a segunda, representa a situação em que tal condição é falsa (disparando a transição t_2). Logo, a representação dos disparos das transições t_1 (Figura 20 (b)) e t_2 (Figura 20 (c)) simula o teste lógico feito no comando condicional, desviando o fluxo da ficha para a esquerda ou para a direita, respectivamente. De fato, conforme mencionado na seção 2.6.1, as RdPC possuem uma característica especial própria que é a de trabalhar com fichas que podem receber valores específicos (instâncias) do problema modelado. Tais valores podem ser testados por uma condição conhecida como *Guard* [23] que está associada a uma dada transição. O disparo dessa transição só ocorrerá se sua *Guard* produzir um valor verdadeiro, conforme mostrado nas Figuras 20 (b) e (c).

Outra potencialidade das RdPC está no fato de permitirem associar aos arcos do modelo um rótulo que atualiza as instâncias das fichas. Esse é o caso do arco de saída

da transição $write(i)$ da Figura 20 (d). Essa Figura representa o comando iterativo, mostrado na Figura 20 (a) (linhas 5 - 6), no qual as iterações ocorrem com i variando de 1 até 5, sendo i incrementado de 1 a cada iteração. A ficha situada no lugar p_1 possui instâncias que são geradas com a atualização da variável i. Enquanto apenas a *Guard* da transição t_2 for verdadeira, a dinâmica da ficha disparará as transições t_2 e $write(i)$ alternadamente, representando as iterações do comando. Ao final, quando a *Guard* da transição t_2 for falsa e a da transição t_1 for verdadeira, a transição t_1 será disparada, modelando o fim das iterações.

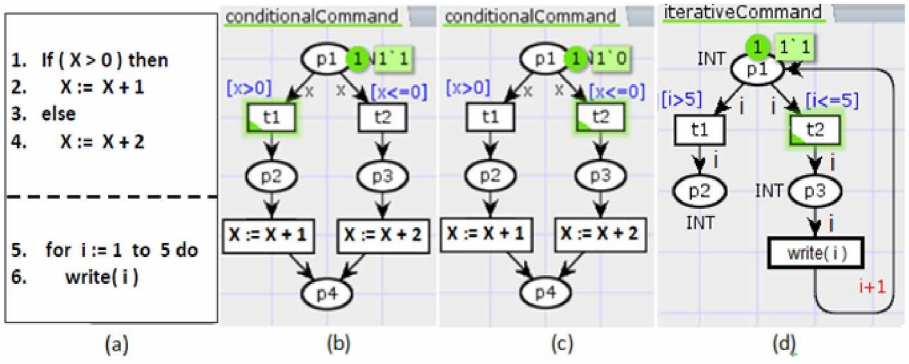


Figura 20 – Modelagem de Comandos Condicional e Iterativo.

Conforme destacado anteriormente, observa-se aqui uma das contribuições do presente trabalho, uma vez que nenhuma outra abordagem nos trabalhos correlatos (tal como em [21]) estão aptas, ou não o fizeram, a apresentarem a dinâmica de desvio de comandos condicionais (conforme explicitado na seção 3).

4.1.3 Comando Recursivo

A Figura 21 exemplifica uma função recursiva denominada de $f1()$. As linhas 2, 3 e 4 representam, respectivamente, o teste para o critério de parada das chamadas recursivas, os retornos recursivos e as chamadas recursivas dessa função. O fluxo de controle de uma função recursiva consiste em duas etapas, nessa ordem: as chamadas recursivas, em que os dados são inseridos (empilhamentos) em uma pilha inicialmente vazia até que um critério de parada seja satisfeito; e os retornos recursivos, nos quais os dados são removidos (desempilhamentos) dessa pilha até esvaziá-la. A ordem de remoção dos dados na pilha deve ser contrária à ordem de suas inserções. As seções 4.1.3.1 e 4.1.3.2 mostram como modelar o fluxo de controle da função recursiva $f1()$, considerando o valor inicial para o seu parâmetro de entrada como sendo $x = 2$.

4.1.3.1 Modelagem da Chamada Recursiva

Na Figura 22 (a), uma ficha com valor $x = 2$ está no lugar p_3, tornando a transição *call* apta ao disparo. Isto representa o fato de que uma chamada recursiva está na iminência

```
1.  function f1 ( int x )
2.    if ( x = 0 ) then
3.      return x
4.    x = x + f1 ( x - 1 )
```

Figura 21 – Exemplo de Função Recursiva.

de acontecer. Quando essa transição é disparada, uma ficha é inserida (empilhamento) no lugar *push* que representa a pilha e outra é inserida no lugar f_1 com o novo valor $x = 1$ (Figura 22 (b)). A *Guard* da transição t_2 é verdadeira ($x <> 0$), fazendo com que essa transição seja disparada (Figura 22 (c)). Novamente, a transição *call* está apta ao disparo. Ao ocorrer o seu disparo, uma nova ficha é inserida no lugar *push* (Figura 22 (d)) simulando assim, dois empilhamentos na pilha.

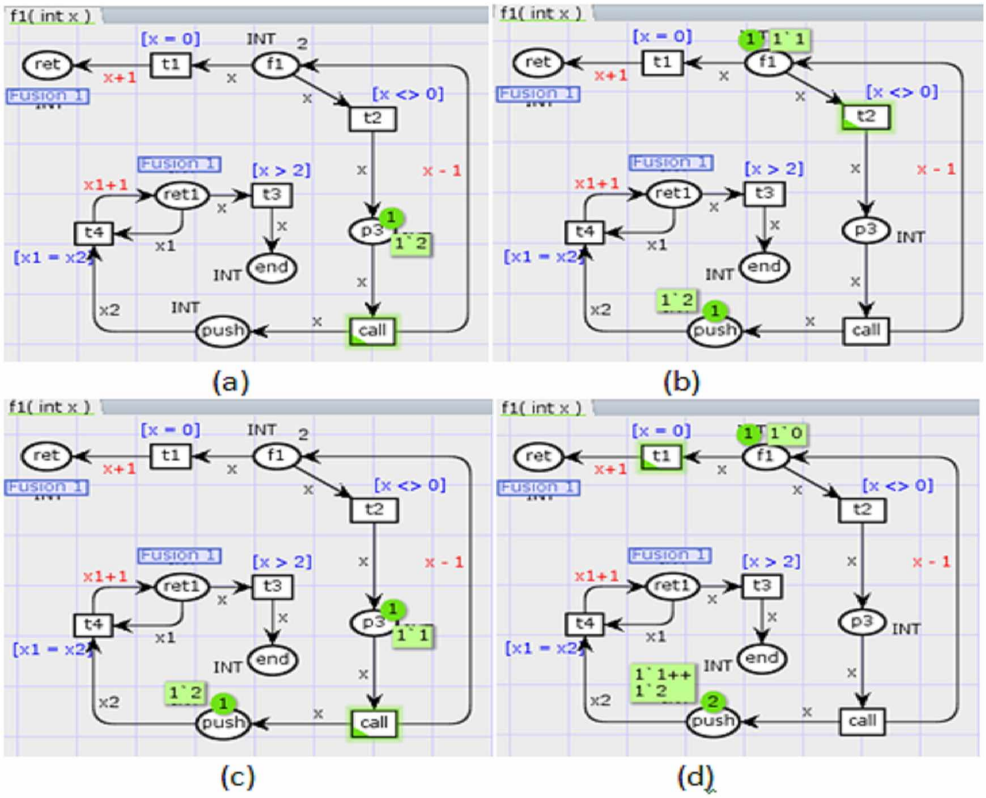


Figura 22 – Modelagem da Chamada Recursiva.

4.1.3.2 Modelagem do Retorno Recursivo

Após a segunda inserção da ficha no lugar *push* (Figura 22 (d)), uma nova ficha com valor $x = 0$ aparece no lugar f_1. Como apenas a *Guard* da transição t_1 é verdadeira, essa transição é disparada colocando uma ficha no lugar *ret* (Figura 23 (a)). Isso modela a ocorrência do critério de parada (linha 2 da Figura 21). Nota-se que a mesma ficha aparece, automaticamente, no lugar *ret*1. Esse fato só é possível porque ambos os lugares *ret* e *ret*1 possuem uma fusão entre si *Fusion1* [23] (o que acontece em *ret* também

acontece em $ret1$). Assim, o fluxo da ficha pode ser desviado de um lugar para outro do modelo sem que haja um *link* explícito entres eles. Isso permitiu simular de forma eficaz no modelo, o desvio do fluxo de dados causado pelos retornos recursivos. Esse recurso existente no CPN Tools foi um dos aspectos motivadores para a escolha dele como ambiente de modelagem e simulação. A Figura 23 (a) mostra ainda que a transição t_4 está apta ao disparo, uma vez que sua *Guard* é verdadeira. Observa-se que existe uma ficha no lugar *push* e uma ficha no lugar $ret1$ cujas instâncias são iguais, tornando o teste lógico da Guard associada à transição t_4 verdadeiro ($x1 = x2$). Quando a transição t_4 é disparada (Figura 23 (b)), a primeira ficha removida (desempilhamento) do lugar *push* é a última ficha inserida nesse lugar durante as chamadas recursivas, conforme esperado. Na sequência, a Figura 23 (c) mostra a remoção da segunda ficha no lugar *push*, esvaziando-o completamente. Por fim, como a *Guard* da transição t_3 é verdadeira, essa transição é disparada (Figura 23 (d)) simulando o fim da recursividade.

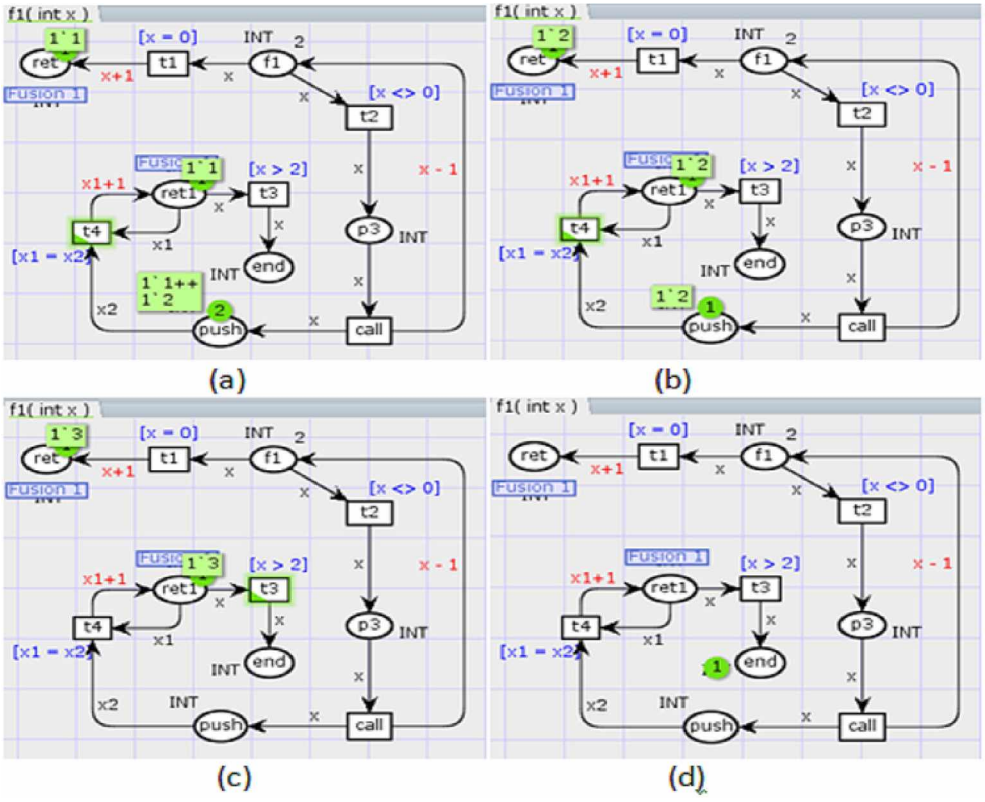


Figura 23 – Modelagem do Retorno Recursivo.

É importante destacar que os lugares de fusão configuram uma contribuição inovadora na modelagem de processos recursivos uma vez que não foram usados em outros trabalhos como em [19] e [20]. O uso desses lugares permite que se represente diversos pontos de desvios em partes distintas do modelo sem comprometer sua apresentação visual de forma didática e legível. De fato, os lugares de fusão reduzem a quantidade de arcos ligados a eles, simplificando a estrutura do modelo como um todo. No modelo de RdP usado por [19], seria inviável a modelagem de um algoritmo fortemente recursivo como

o Minimax, por exemplo, que apresenta duas chamadas recursivas e vários comandos de retornos recursivos. Como será mostrado nas próximas seções, essa limitação não foi verificada no modelo proposto nesta tese para modelar o algoritmo Minimax.

4.2 Modelagem do Algoritmo Minimax

A semântica usada para modelar a estrutura do algoritmo Minimax é a mesma apresentada na seção 4.1. A Figura 24 mostra o pseudo-código desse algoritmo, cuja estrutura é composta por três partes funcionais que verificam se o nó que está sendo processado é um nó-folha (*VerifyLeaf*), um nó que está no nível de maximização (*VerifyMax*), ou um nó que está no nível de minimização (*VerifyMin*).

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

Figura 24 – Pseudo-Código do Minimax.

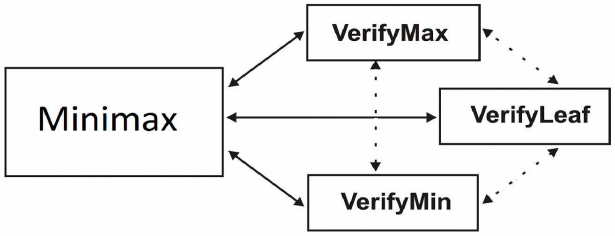


Figura 25 – Visão Macro do Modelo do Algoritmo Minimax.

Na Figura 25 tem-se uma visão macro da modelagem informal do algoritmo, em que cada parte funcional é representada por um módulo. O módulo *Minimax* representa a

rede principal situada no nível hierárquico mais abstrato do modelo. Os módulos restantes correspondem às sub-redes do modelo associadas às partes *VerifyLeaf*, *VerifyMax* e *VerifyMin*. As setas contínuas representam a comunicação existente entre a rede principal e as sub-redes, ao passo que as setas tracejadas representam a comunicação existente entre as sub-redes que estão no mesmo nível hierárquico. A necessidade de modelar essas comunicações foi a motivação principal para o uso de uma RdPCH em conjunto com o CPN Tools. De fato, o recurso da hierarquia presente na RdPCH, aliado à possibilidade de se usar lugares de fusão, transições abstratas e *sockets* presentes no ambiente CPN Tools, ofereceu um suporte técnico robusto para a implementação da abordagem proposta nesta tese. Enquanto as transições abstratas definem o fluxo das fichas, partindo da rede principal (nível mais abstrato) para as sub-redes (nível menos abstrato), os lugares de fusão e os *sockets* se responsabilizam em definir o fluxo dessas fichas no sentido contrário.

Na Figura 26 é apresentado o modelo de RdPCH que representa o algoritmo Minimax. Nota-se que as três transições abstratas apresentadas com bordas duplas situadas na rede principal *Minimax* representam o *link* para as sub-redes $VerifyLeaf$, $VerifyMax$ e $VerifyMin$. É intuitivo notar na Figura 26 que a abordagem hierárquica das RdPCH confere uma modelagem mais didática facilitando o entendimento e a manutenção do modelo, uma vez que as partes funcionais estão bem definidas.

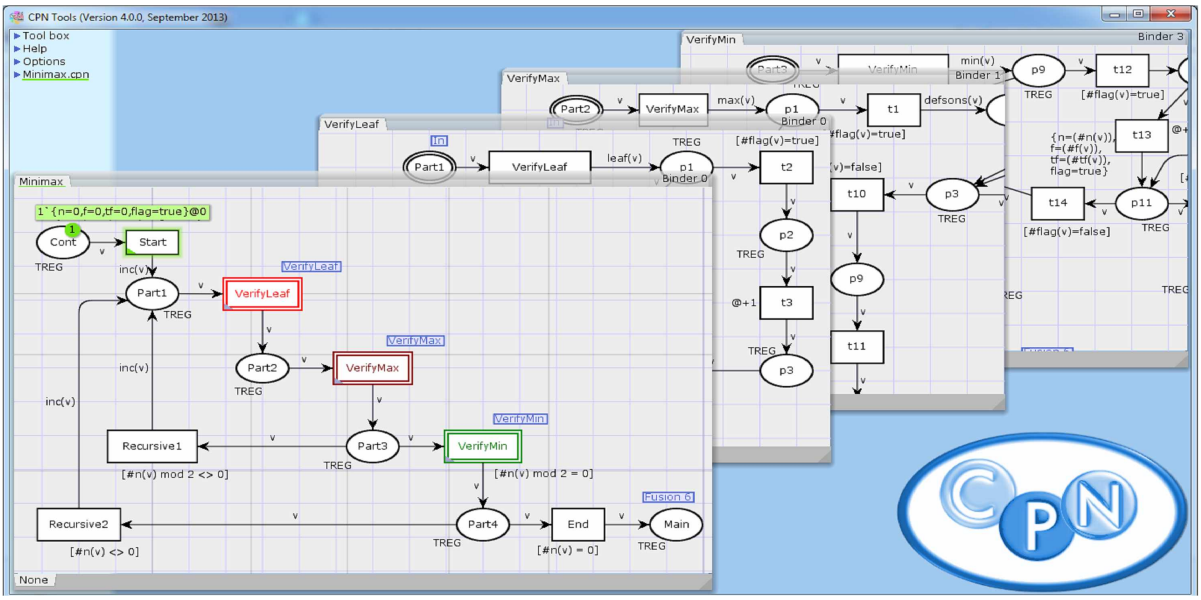


Figura 26 – Modelando o Algoritmo Minimax.

Além disso, na Figura 26, observa-se que a marcação inicial na rede principal *Minimax* é dada pela presença de uma ficha no lugar *Cont*, o que significa que a transição *Start* pode ser disparada, simulando o início da busca em uma árvore. Posteriormente, quando aparecer uma ficha no lugar *Main*, será o fim da busca na árvore explorada. Em todo o modelo, os arcos que ligam os lugares às transições e as transições aos lugares possuem um rótulo definido, ou pela variável local v, ou por uma função criada para algum pro-

pósito. A variável v é responsável por carregar as instâncias das fichas à medida que elas fluem pelo modelo. Sabendo que em uma RdPCH essas instâncias são constantemente usadas, impactando diretamente na dinâmica do modelo, o papel da variável v torna-se fundamental. Essa variável foi declarada, inicialmente, com o tipo abstrato REG, cuja estrutura é dada por:

$$REG = \left\{ \; n, \quad f, \quad t_f, \quad flag \; \right\} \tag{3}$$

onde,

- n é do tipo *Integer* que representa o nível atual da árvore que está sendo explorado;
- f é do tipo *Integer* que representa o número de nós filhos do nó atual que não foram processados ainda;
- t_f é do tipo *Integer* que representa o fator de ramificação do nó atual;
- $flag$ é do tipo *Boolean* que pode ser usada na política de controle dos comandos condicionais e iterativos.

Uma vez que o alvo dessa etapa é o cálculo da função de complexidade do tempo de execução do algoritmo Minimax e baseando-se nas simulações do modelo, foi necessária a inclusão de um operador de tempo. Portanto, o tipo REG foi estendido para o tipo abstrato $TREG$, cuja estrutura é dada por:

$$TREG = \quad REG \quad Timed \tag{4}$$

Observa-se que $TREG$ está associada a todos os lugares do modelo, determinando que essa é a cor das fichas que fluirão por esses lugares. A Figura 27 mostra a declaração dos tipos REG e $TREG$ feitas no ambiente CPN Tools.

4.2.1 Modelagem da Sub-rede $VerifyLeaf$

A Figura 28 mostra a sub-rede $VerifyLeaf$ que representa a primeira parte funcional do pseudo-código da Figura 24. O comando condicional que aparece na linha 2 do algoritmo é representado pelas transições t_1 e t_2 da Figura 28. O fluxo de controle desse comando condicional, o qual faz o teste lógico para saber se o nó atual é um nó-folha ou não, é modelado pela combinação da função $leaf(v)$, que rotula o arco de entrada do lugar p_1 com as *Guard* associadas às transições t_1 e t_2.

A Figura 27 mostra como a função $leaf(v)$ foi criada. É importante destacar que o modelo criado nesta etapa considera as explorações feitas em árvores em que todas as ramificações possuem a mesma profundidade. Além disso, os nós das árvores possuem uma quantidade variada de nós filhos (árvores não-uniformes). Assim, nota-se que a

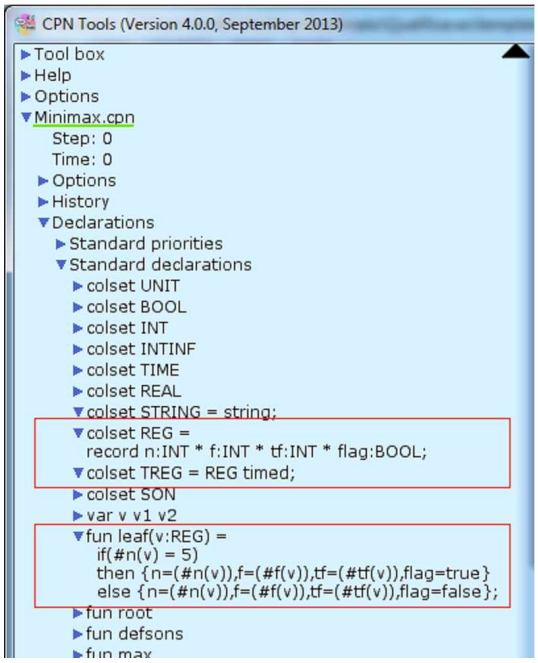


Figura 27 – Exemplos de Declarações no CPN Tools.

função $leaf(v)$ verifica o valor da variável n para saber se o nó é ou não um nó-folha. Em caso afirmativo, a função $leaf(v)$ faz $flag = true$ e a transição t_2 é disparada, caso contrário, ela faz $flag = false$ e a transição t_1 é disparada.

Para o caso de haver o disparo da transição t_2 (verificação de nó-folha), de acordo com o pseudo-código da Figura 24 uma função é chamada e um retorno recursivo é realizado (linha 2). Essas duas execuções são representadas pelos disparos sequenciais das transições t_3 e t_4 da Figura 28.

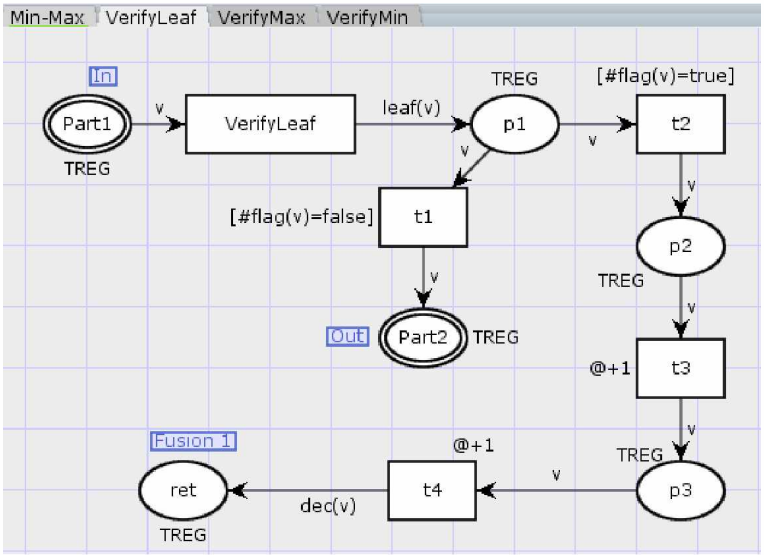


Figura 28 – Sub-rede VerifyLeaf.

É importante notar que cada uma das transições que modelam um comando do al-

goritmo possui um operador de tempo com custo unitário @ + 1 associado a elas [23], como no caso das transições t_3 e t_4 da Figura 28. Esse recurso possibilita a contabilização dos tempos de execução de cada comando modelado. Assim, cada vez que uma dessas transições é disparada, o CPN Tools calcula seu custo temporal em um indicador de tempo situado no próprio ambiente de simulação. Ao final de uma simulação completa de um cenário, esse indicador tem como valor a soma das unidades de tempo de todas as transições que possuem um operador de tempo associado a elas. Variando-se os dados de entrada no modelo que neste caso são as profundidades máximas das árvores e seus fatores de ramificações médios, ao final de diversas simulações de cenários distintos, obtém-se o comportamento da função de complexidade de tempo. Tal estratégia se baseia no método de análise do tempo de execução conhecido como "Notação Assintótica" citado no capítulo 1 desta tese.

Para exemplificar, a Figura 29 mostra como é feita a contabilização dos tempos de execução de cada comando durante uma busca. Observa-se que o aprofundamento máximo da árvore com $n = 5$ foi estipulado na função $leaf(v)$ (Figura 27). Na Figura 29(a), quando a busca atinge um nó-folha, a transição t_2 estará habilitada para o disparo. Ao ser disparada, a ficha chega no lugar p_2, habilitando a transição t_3 ao próximo disparo, conforme mostrado na Figura 29(b). Deve-se notar que, no lado esquerdo da Figura 29(b), a contabilização do tempo total gasto na busca até o momento é de oito unidades de tempo. Após uma unidade de tempo que é a temporização associada à transição t_3, essa transição é disparada e a ficha chega ao lugar p_3. A Figura 29(c) mostra que o tempo total gasto pela busca passou para nove unidades de tempo. Por fim, a transição t_4 é habilitada ao disparo que ocorrerá após uma unidade de tempo (temporização associada à transição t_4). De fato, a Figura 29(d) mostra que, ao disparar t_4, o tempo total gasto na busca passou a ser de dez unidades de tempo. Assim, o processo de contabilização do tempo continua durante todo o fluxo das fichas pelo modelo até o processo de busca ser finalizado.

Analisando os lugares $Part1$ e $part2$ da Figura 28, nota-se que ambos possuem *sockets* de entrada (In) e de saída (Out), definindo, respectivamente, o fluxo de entrada e de saída das fichas na sub-rede $VerifyLeaf$. Além disso, como a transição t_4 modela o comando de retorno recursivo, o seu lugar de saída possui uma fusão $Fusion1$ com um lugar situado em outra parte do modelo. Isso representa de forma correta o desvio do fluxo de dados que um comando de retorno recursivo pode provocar em um algoritmo.

4.2.2 Modelagem das Sub-redes $VerifyMax/VerifyMin$

Analisando as três partes funcionais da Figura 24, nota-se que $VerifyMax$ e $VerifyMin$ são semelhantes estruturalmente, porém são mais complexas do que $VerifyLeaf$. Isso fez com que as sub-redes $VerifyMax$ e $VerifyMin$ sejam estruturalmente mais complexas.

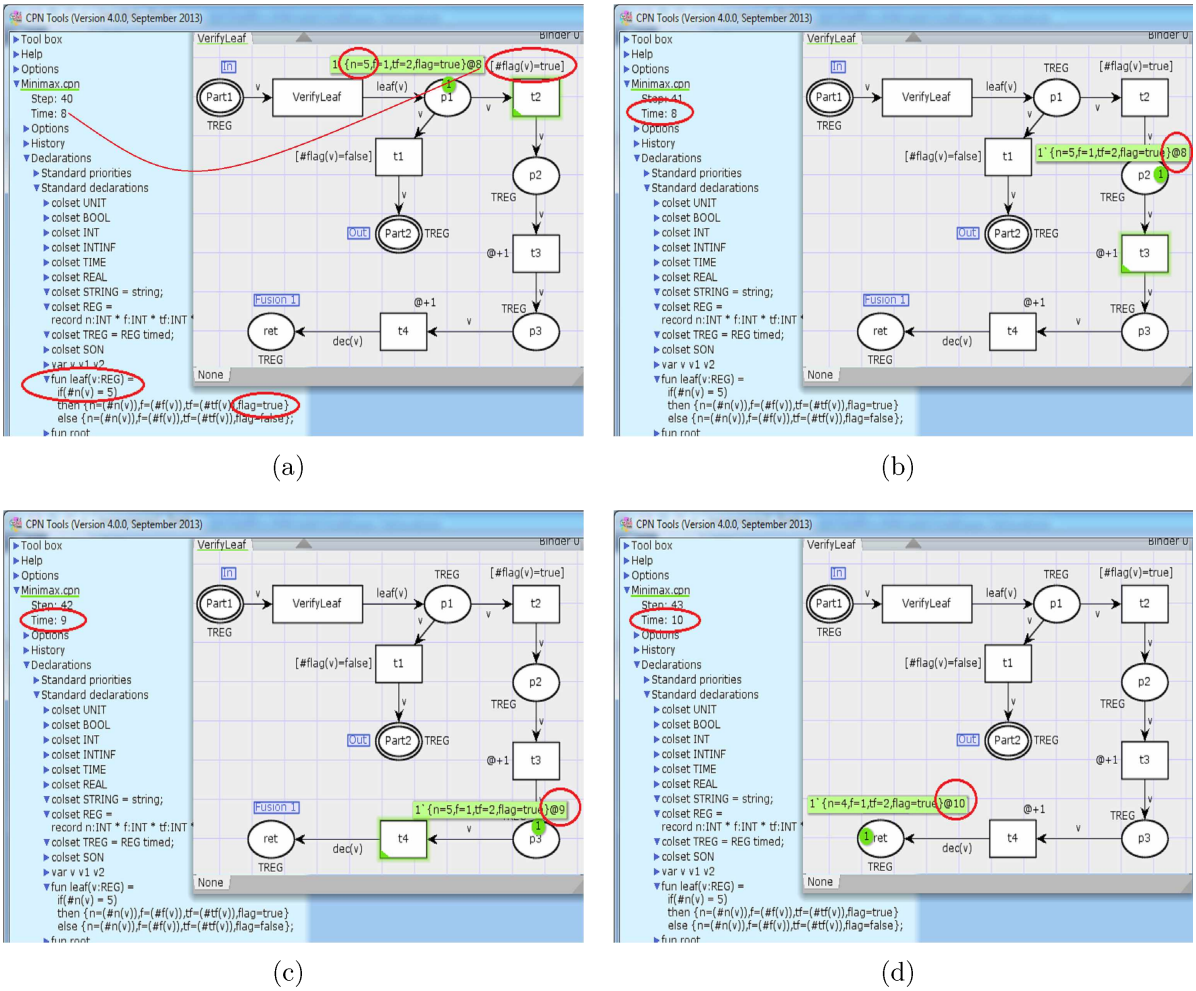


Figura 29 – Exemplo de Contabilização dos Tempos de Execução

De fato, sem considerar o detalhamento das informações, na sub-rede da Figura 30 há uma quantidade maior de elementos do que na sub-rede da Figura 28.

A parte $VerifyMax$ da Figura 24 possui um grupo de comandos que pertence ao escopo de um comando iterativo (linhas 5 - 9). Na sub-rede da Figura 30, os elementos que representam tais comandos estão destacados em azul. O fluxo de controle do comando iterativo exigiu no modelo a criação de três funções de controle, sem as quais não seria possível simular automaticamente os cenários completos existentes. Isso se deve ao fato de que o algoritmo Minimax sempre explora todas as ramificações da árvore. Tais ramificações são definidas pela quantidade de nós filhos (fator de ramificação) que cada nó atual possui. Sabe-se que no algoritmo, o fator de ramificação determina o número de iterações do comando iterativo, cujo fluxo de controle deve ser representado no modelo. A Figura 31 mostra um exemplo de declaração das três funções de controle: $defsons(v)$, $decsons(v)$ e $verifyLoop(v)$.

Em tal exemplo, a função de controle $defsons(v)$ é composta por uma distribuição uniforme com média igual a 2, fazendo com que sejam gerados valores aleatoriamente, no caso, valores inteiros entre 1 e 3. Ou seja, os nós da árvore poderão ter um, dois ou três

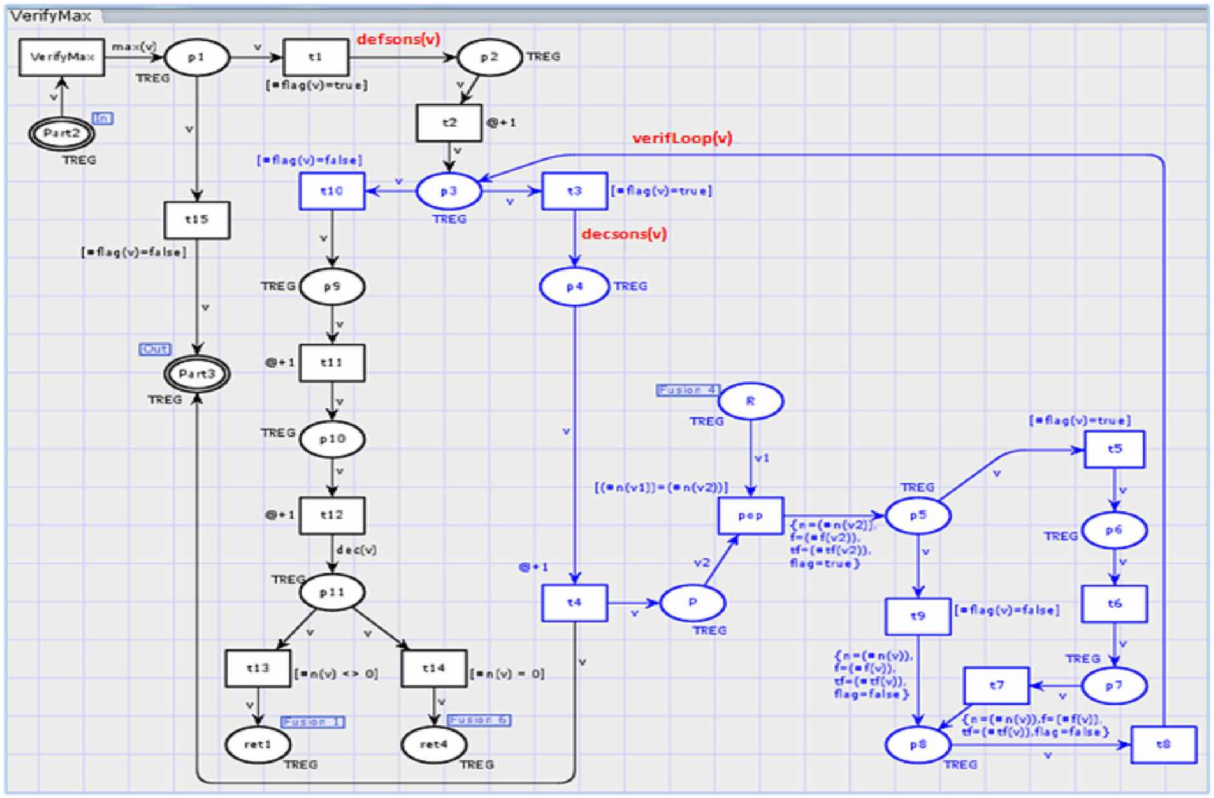


Figura 30 – Modelagem da Sub-rede VerifyMax.

filhos. Esse valor é atribuído às variáveis tf e f, relacionadas ao controle do número de nós filhos a serem processados. Além disso, verifica-se que a função $defsons(v)$ foi usada para etiquetar o arco de saída da transição t_1 da Figura 30, cujo disparo ocorrerá toda vez que um novo nó n for processado. A escolha de uma distribuição uniforme para definir a quantidade de filhos de um nó reflete o fato de que a escolha aleatória dessa quantidade possui a mesma probabilidade para qualquer resultado obtido, ou seja, para o intervalo de valores inteiros entre 1 e 3, a chance de um nó possuir um, dois ou três filhos é a mesma. O intervalo escolhido poderia ser mais amplo o que resultaria em um valor médio maior para os nós em relação às suas quantidades de nós filhos, portanto, é uma escolha que depende do quão espalhada se quer que uma árvore seja.

Sempre que ocorrer uma nova iteração no comando iterativo (linha 5) da Figura 24, a transição t_3 da Figura 30 é disparada. Quando isso acontece, um nó filho do nó atual acaba de ser processado. Logo, é necessário que a contagem de nós filhos do nó atual que ainda não foram processados seja decrementada de um. A função $decsons(v)$, que rotula o arco de saída da transição t_3, realiza essa ação no modelo, conforme mostra a Figura 31.

Para que o critério de parada do comando iterativo seja atingido, é necessário que se faça um teste lógico antes de cada iteração nova. Esse é o papel da função $verifLoop(v)$, a qual verifica se ainda há ou não nós filhos não processados do nó atual. De acordo com

a Figura 31, quando a variável f for igual a zero, isso representará o fim das iterações do comando iterativo e, portanto, a função $verifLoop(v)$ retornará um valor $flag = false$. Como a *Guard* da transição t_{10} da Figura 30 testa exatamente se $flag = false$, isso faz com que o critério de parada seja atingido e a transição t_{10} seja disparada.

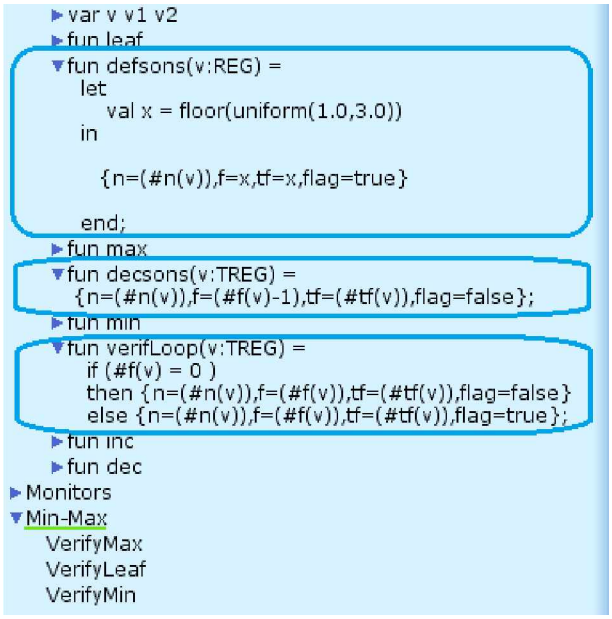


Figura 31 – Declaração das Funções de Controle do Comando Iterativo.

Saliente-se que, diferentemente da presente proposta, as abordagens apresentadas nos trabalhos correlatos (capítulo 3) não estão aptas a permitir simulações automáticas dos modelos criados, uma vez que não permitem a criação de funções de controle (como $defsons(v)$) indispensáveis a tal fim. Logo, a habilidade para efetuar automaticamente as simulações dos modelos produzidos para os algoritmos representa uma outra contribuição do presente trabalho.

4.3 Simulações Automáticas do Modelo: Avaliando a Função de Complexidade do Minimax

Antes de apresentar o processo de simulação automática do modelo obtido, é importante destacar que:

- a adequação da abordagem proposta ao cálculo da função de complexidade do Minimax será validada por meio da comparação entre os comportamentos das curvas obtidas por meio da simulação automática do modelo de tal algoritmo e via análise teórica tradicional, levando-se em conta apenas o comportamento dessas curvas;
- conforme explicado na seção 4.2.1, o custo temporal de cada comando modelado foi fixado com o valor de uma unidade de tempo;

- como contribuição adicional, as simulações aqui efetuadas envolverão o algoritmo Minimax modelado operando com dados não-uniformes. Tal aspecto é representado no modelo através do uso de valores obtidos por meio de funções de distribuição estocástica para o fator de ramificação das árvores envolvidas;
- a função de complexidade do tempo de execução do algoritmo Minimax conhecida na literatura é dada por $O(b^m)$, sendo b o fator de ramificação médio da árvore e m a sua profundidade máxima.

A fim de gerar a curva da função de complexidade do Minimax, os cenários foram replicados trinta vezes. Cada cenário se caracteriza pela simulação de uma árvore não-uniforme [14], ou seja, uma árvore que tenha uma dada profundidade máxima e cujos nós podem conter fatores de ramificação distintos entre si, sendo os mesmos gerados aleatoriamente pela função de controle $defsons(v)$. Além disso, a raiz de cada árvore está no nível 1.

Como ao longo de cada cenário simulado o fator de ramificação varia, a fim de possibilitar o controle automático do fluxo dos comandos iterativos que são delimitados por tais fatores, são usadas as funções de controle $decsons(v)$ e $verifLoop(v)$. O resultado de cada cenário simulado representa o tempo de execução do algoritmo naquele cenário.

A título de diversificação dos testes, foram efetuados dois conjuntos de simulações: um primeiro que prioriza a profundidade da árvore em detrimento do fator de ramificação e um segundo que faz o contrário. Particularmente, nos testes do primeiro conjunto o fator de ramificação médio das árvores é igual a 2 e as profundidades variam entre $\{3, 5, 7, 9, 11, 13, 15\}$ níveis máximos. Consequentemente, o i-ésimo nível corresponde ao $(i-1)$-ésimo aprofundamento da árvore. Já nos testes do segundo conjunto foi usado um fator de ramificação médio igual a 5 e as profundidades variando entre $\{3, 4, 5, 6, 7, 8, 9\}$ níveis máximos.

As Figuras 32, 33 e 34 se referem ao primeiro conjunto de simulações. Já as Figuras 35, 36 e 37 se referem ao segundo conjunto de simulações. As sub-seções a seguir apresentam esses conjuntos.

4.3.1 Primeiro Conjunto de Simulações

A Figura 32 mostra os resultados das 210 simulações efetuadas considerando o fator de ramificação médio igual a 2. Cada linha da tabela representa sete cenários simulados, cada um deles referente às profundidades com $3, 5, 7, 9, 11, 13$ e 15 níveis máximos.

A partir dos resultados experimentais obtidos pela abordagem proposta mostrados na última linha da Figura 32 (valores médios dos tempos de execução em unidade de tempo), geraram-se a tabela (Figura 33 (a)) e a curva (Figura 34 (a)) que mostram a variação do tempo de execução do Minimax em função da profundidade árvore. A título de validação desses resultados, os mesmos foram comparados com os valores obtidos a

CPNTools (tempos de execução em unidade de tempo)						
3 níveis	5 níveis	7 níveis	9 níveis	11 níveis	13 níveis	15 níveis
13	76	323	2747	5293	37673	135793
23	51	124	373	15073	9861	28428
30	18	309	4640	18953	48556	55997
26	143	1165	1686	1328	83322	127685
33	86	729	892	9423	31167	204440
13	106	537	1964	4995	58515	151158
30	172	412	339	2799	22268	191437
24	152	422	1603	1758	22359	165291
29	69	398	1895	1206	12477	208683
20	121	237	2817	14055	47951	192220
29	63	372	2393	1491	19028	40623
33	118	421	1115	16119	35573	199599
42	81	94	2800	6933	39561	271363
36	236	546	605	19285	6986	56077
36	28	320	1197	9903	38622	177473
23	134	584	596	2294	4652	90598
10	55	269	2203	8307	69252	110567
10	138	554	256	2536	10806	79494
16	241	206	479	6920	40046	86241
36	45	460	779	9627	40444	108226
16	205	223	1404	3742	36713	18487
33	34	143	4298	3201	56477	174119
29	153	391	3065	8800	20235	93707
23	170	242	1838	13821	31466	79498
23	75	537	554	5570	17627	98295
30	110	812	274	7391	43933	166510
24	193	353	2388	1577	6810	157766
23	96	593	3494	4312	17909	83277
23	170	299	1619	19312	22453	152236
23	119	149	2680	2301	35358	126954
25.30	**115.27**	**407.47**	**1766.43**	**7610.83**	**32270.00**	**127741.40**

Figura 32 – Primeiro Conjunto: Tempos de Execução Médios Obtidos Experimentalmente.

partir da função de complexidade $O(b^m)$ do Minimax (calculada por meio da análise teórica), onde $b = 2$ e m assume valores variando entre $\{2, 4, 6, 8, 10, 12, 14\}$ profundidades máximas. Para tanto, os valores obtidos na análise teórica foram representados na tabela da Figura 33 (b) e plotados na curva da Figura 34 (b). A comparação do comportamento das curvas obtidas a partir da abordagem proposta (Figura 34 (a)) e da análise teórica (Figura 34 (b)) comprova a adequação da ferramenta aqui proposta. De fato, ambas as curvas são regressões exponenciais cujos fatores de correlação são extremamente próximos uns dos outros, tendendo ao valor unitário. Nota-se que, fator de correlação obtido experimentalmente foi de 0.9475, o fator de correlação obtido teoricamente foi de 0.9487.

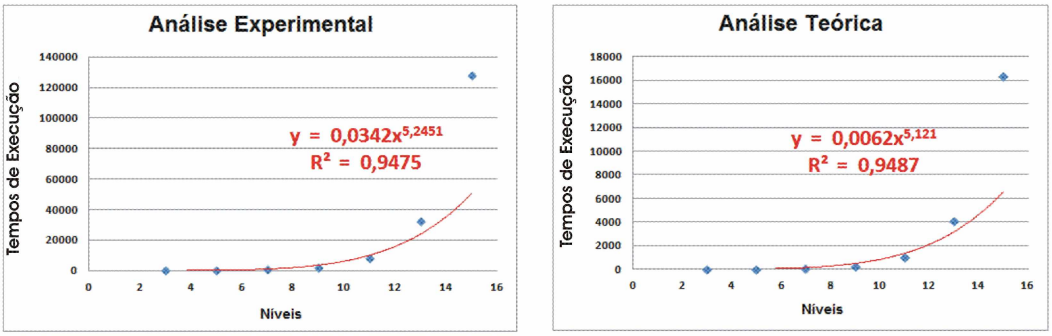
Análise Experimental	
Níveis	Tempos de Execução
3	25.30
5	115.27
7	407.47
9	1766.43
11	7610.81
13	32270.00
15	127741.40

(a)

Análise Teórica	
Níveis	$O(b^m)$
3	4
5	16
7	64
9	256
11	1024
13	4096
15	16384

(b)

Figura 33 – Primeiro Conjunto: Comparativo dos Tempos de Execução Experimental e Teórico do Minimax.



(a) (b)

Figura 34 – Primeiro Conjunto: Comparativo das Funções de Complexidade Experimental e Teórica do Minimax.

4.3.2 Segundo Conjunto de Simulações

A Figura 35 mostra os resultados das 210 simulações efetuadas com fator de ramificação médio igual a 5. Cada linha da tabela representa sete cenários simulados, cada um deles referente às profundidades com $3, 4, 5, 6, 7, 8$ e 9 níveis máximos.

A partir dos resultados experimentais obtidos pela abordagem proposta mostrados na última linha da Figura 35 (valores médios dos tempos de execução), geraram-se a tabela (Figura 36 (a)) e a curva (Figura 37 (a)) que mostram a variação do tempo de execução do Minimax em função da profundidade árvore. A título de validação desses resultados, os mesmos foram comparados com os valores obtidos a partir da função de complexidade $O(b^m)$ do Minimax (calculada por meio da análise teórica), onde $b = 5$ e m assume valores variando entre $\{2, 3, 4, 5, 6, 7, 8\}$ profundidades máximas. Para tanto, os valores obtidos na análise teórica foram representados na tabela da Figura 36 (b) e plotados na curva da Figura 37 (b). A comparação do comportamento das curvas obtidas a partir da abordagem proposta (Figura 37 (a)) e da análise teórica (Figura 37 (b)) comprova a adequação da ferramenta aqui proposta. De fato, ambas as curvas são regressões exponenciais cujos fatores de correlação são extremamente próximos uns dos outros, tendendo ao valor uni-

CPNTools (tempos de execução em unidades de tempo)						
3 níveis	4 níveis	5 níveis	6 níveis	7 níveis	8 níveis	9 níveis
161	528	2462	10878	21638	3401	140230
75	665	2374	13034	5029	280949	680262
101	253	2436	5395	37873	204670	554883
100	358	412	15392	30397	258751	1010719
47	587	2351	3476	64608	42382	632397
182	506	851	5028	50683	228463	590501
87	194	778	3945	19219	254879	603206
45	215	1716	943	11484	34082	777105
152	181	2547	10698	13063	251305	102570
72	192	322	3802	26601	50376	1580221
85	42	1533	8005	69819	188469	1152700
47	96	2611	4150	37389	51266	165518
38	412	1913	16269	40173	282492	1196742
10	640	2332	3322	61219	47817	330537
16	673	1954	730	46595	267750	706103
60	479	474	3939	10024	220444	497972
95	119	3327	3387	1240	166993	414852
44	519	647	5575	54072	69702	330176
47	156	887	11739	8756	36228	624167
38	243	2599	4754	28788	213252	294238
32	197	330	8977	36738	54904	957954
90	492	891	5755	45449	236691	504812
25	243	1295	10096	21813	147030	964261
31	473	730	5659	58675	138868	926427
114	570	1280	827	36751	126131	1143554
140	426	2177	2559	13051	292065	439486
113	600	194	13684	11995	122194	894640
167	241	1262	4088	10382	252874	50148
35	228	1430	2336	76011	107858	1027318
140	469	2725	9226	22499	220810	763492
79.63	**366.57**	**1561.33**	**6588.93**	**32401.13**	**161769.87**	**668573.03**

Figura 35 – Segundo Conjunto: Tempos de Execução Médios Obtidos Experimentalmente.

tário. Nota-se que o fator de correlação obtido experimentalmente foi de 0.9727, o fator de correlação obtido teoricamente foi de 0.9754.

4.4 Considerações Relativas ao Capítulo

Este capítulo mostrou como é possível fazer a avaliação da função de complexidade do tempo de execução de um algoritmo serial por meio de uma análise empírica baseada na

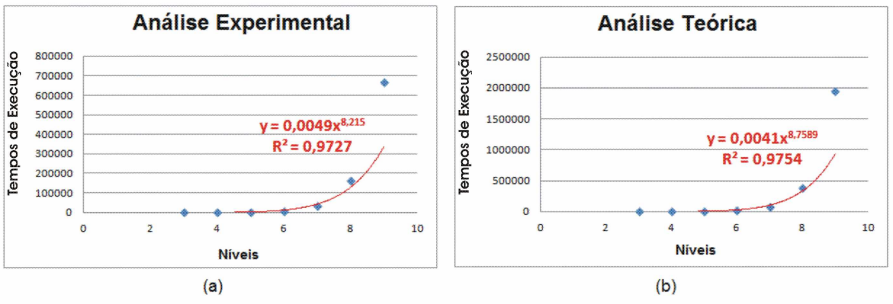
Análise Experimental	
Níveis	Tempos de Execução
3	79.63
4	366.57
5	1561.33
6	6588.93
7	32401.13
8	161769.87
9	668573.03

(a)

Análise Teórica	
Níveis	$O(b^m)$
3	125
4	625
5	3125
6	15625
7	78125
8	390625
9	1953125

(b)

Figura 36 – Segundo Conjunto: Comparativo dos Tempos de Execução Experimental e Teórico do Minimax.



(a) (b)

Figura 37 – Segundo Conjunto: Comparativo das Funções de Complexidade Experimental e Teórica do Minimax.

simulação automática de modelos formais. Com a realização desta etapa, pôde-se concluir que:

- a RdPCH foi capaz de modelar tanto a estrutura quanto as políticas de controle dos comandos de um algoritmo composto por diversos comandos que provocam desvios intensos nos fluxos de dados durante a sua execução. Isso foi possível graças à possibilidade de se agregar ao modelo as potencialidades existentes na programação funcional. A possibilidade de acompanhar a evolução dinâmica das fichas usando um modelo de RdP com o uso das *Guards*, do operador de tempo, dos *sockets*, das transições abstratas e dos lugares de fusão presentes nas RdPCH e de recursos como a declaração de tipos abstratos de dados e de funções de controle presentes na programação funcional *Standard ML*, propiciaram simulações automáticas eficazes dos cenários. Tais recursos não estão presentes em ferramentas como Autômatos, Grafos ou UML, conforme citado na seção 1.1;

- o uso das hierarquias por meio da RdPCH fez com que tanto o processo de modelagem quanto a análise do comportamento dinâmico do modelo se tornassem mais fáceis de serem realizados em virtude da caracterização das partes funcionais do algoritmo na própria estrutura do modelo;
- a corretude da abordagem sugerida nesta etapa foi comprovada uma vez que as curvas da função de complexidade dos tempos de execução obtidos experimentalmente e teoricamente mostraram-se muito próximas. Isso produziu valores estatísticos bem aproximados, como os fatores de correlação de ambos os casos que tenderam ao valor unitário;
- o ambiente CPN Tools possibilitou a criação visual do modelo e a realização das simulações automáticas de cenários completos, proporcionando uma análise empírica totalmente isenta da necessidade de implementação do algoritmo modelado.

Capítulo 5

Avaliação dos Tempos de Execução Reais dos Algoritmos por meio das RdPCH

Este capítulo detalha a segunda etapa de desenvolvimento desta tese, a qual consiste na avaliação dos tempos de execução reais dos algoritmos por meio de simulações automáticas de modelos de RdPCH usando o ambiente CPN Tools. A principal motivação desta etapa é a validação da abordagem proposta para algoritmos com complexas estruturas de comandos. O algoritmo usado como caso de teste é o Alpha-Beta, que apresenta a mesma estrutura de comandos do algoritmo de busca serial Minimax, acrescida dos comandos responsáveis pela ocorrência das podas durante a exploração em uma árvore de busca. A necessidade de se representar esses comandos adicionais no modelo propiciou um desafio maior do que aquele apresentado na primeira etapa desta tese uma vez que foi necessário o uso dos seguintes recursos disponíveis no CPN Tools: os operadores de tempo com custo real (e não com custos unitários como foi feito na primeira etapa) que foram associados às transições que modelam cada comando do algoritmo e as funções de controle que utilizam tanto valores de probabilidades quanto funções de distribuição para representar os fatores de ramificação e as podas existentes.

Diferentemente da primeira etapa, os parâmetros adicionais do modelo (operadores de tempo real, probabilidades e funções de distribuição) precisaram ser bem refinados de modo a representarem de forma eficaz os valores reais dos dados que impactam diretamente no tempo de execução real do algoritmo. Com isto, o ajuste de tais parâmetros foi produzido realizando-se diversas execuções do algoritmo implementado e obtendo-se os valores médios: dos tempos de execução reais de cada comando, dos fatores de ramificação dos nós da árvore explorada e das ocorrências de podas produzidas pelo algoritmo.

Por fim, com a obtenção do modelo e com o ajuste de seus parâmetros, comparou-se a curva obtida por meio de suas simulações automáticas dos modelos com a curva real dos tempos de execução do algoritmo e não com a sua curva teórica, como foi feito na

primeira etapa.

5.1 Modelagem do Algoritmo Alpha-Beta

A Figura 38 mostra o pseudo-código do algoritmo de busca serial Alpha-Beta, cuja estrutura é composta por quatro partes funcionais que verificam se o nó que está sendo processado é um nó-folha (*VerifyLeaf*), um nó-raiz (*VerifyRoot*), um nó que está no nível de maximização (*VerifyMax*), ou um nó que está no nível de minimização (*VerifyMin*).

```
1.  fun alfaBeta( n:node, depth:int, min:int, max:int, bestmove:move ) :float
2.    if( leaf(n) ) then
3.      besteval := evaluate(n)                                          VerifyLeaf
4.      store( n, besteval, bestmove, depth, hashExact )
5.      return besteval
6.    if( root(n) ) then
7.      if( not isEmpty(bestmove) ) then                                 VerifyRoot
8.        setChildrenOrder( n, bestmove )
9.    if( n is a max node ) then
10.     besteval := min
11.     for each child of n
12.       if( retrieve( child, besteval, bestmove, depth+1, parentIsMaxNode ) )
13.         then v := besteval
14.         else v := alfaBeta( child, depth+1, besteval, max, bestmove )
15.       if( v > besteval ) then                                        VerifyMax
16.         besteval := v
17.         thebest := bestmove
18.       if( besteval >= max ) then
19.         store( child, besteval, bestmove, depth, hashAtLeast )
20.         return besteval
21.     bestmove := thebest
22.     store( n, besteval, bestmove, depth, hashExact )
23.     return besteval
24.   if( n is a min node ) then
25.     besteval := max
26.     for each child of n
27.       if( retrieve( child, besteval, bestmove, depth+1, parentIsMinNode ) )
28.         then v := besteval
29.         else v := alfaBeta( child, depth+1, besteval, min, bestmove )
30.       if( v < besteval ) then
31.         besteval := v                                                VerifyMin
32.         thebest := bestmove
33.       if( besteval <= min ) then
34.         store( child, besteval, bestmove, depth, hashAtMost )
35.         return besteval
36.     bestmove := thebest
37.     store( n, besteval, bestmove, depth, hashExact )
38.     return besteval
```

Figura 38 – Pseudo-Código do Alpha-Beta.

Na Figura 39, é mostrado o modelo de RdPCH que representa o algoritmo Alpha-Beta. Nota-se que as quatro transições abstratas apresentadas com bordas duplas situadas na rede principal *AlphaBeta*, representam o *link* para as sub-redes $VerifyLeaf$, $VerifyRoot$, $VerifyMax$ e $VerifyMin$.

5.1.1 Modelagem dos Tempos de Execução Reais dos Comandos

Para detalhar a modelagem dos tempos reais de execução dos comandos do algoritmo Alpha-Beta, a Figura 40 mostra a sub-rede $VerifyLeaf$. Nota-se que as transições t_1 e t_2 modelam os caminhos alternativos definidos pelo fluxo de controle existente no comando condicional da parte funcional $VerifyLeaf$ do algoritmo Alpha-Beta (linha 2 da Figura

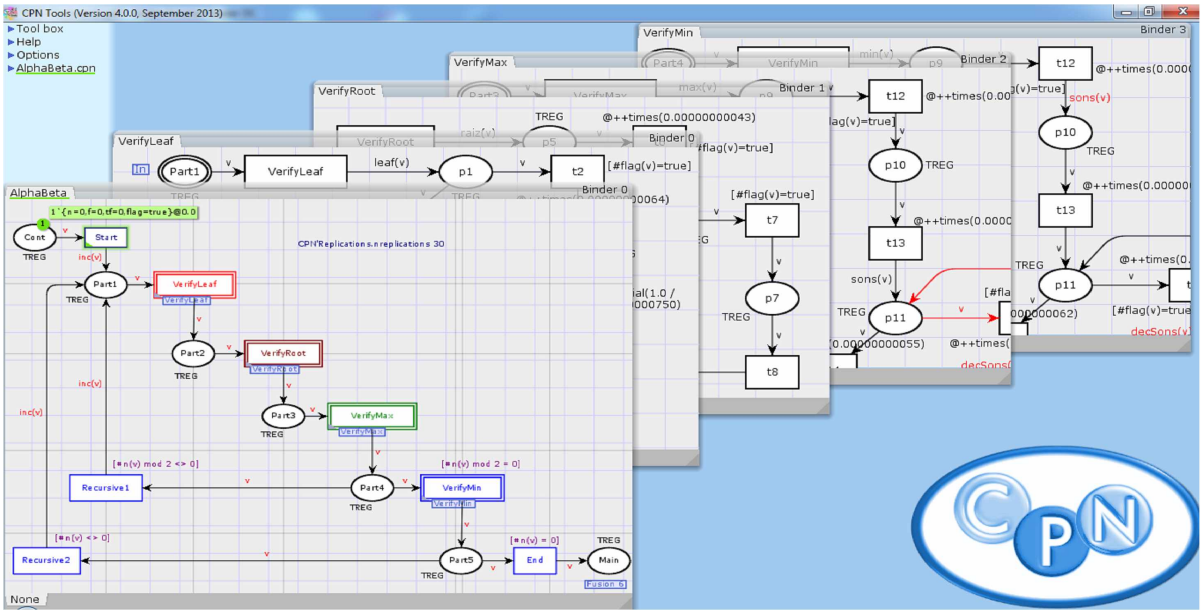


Figura 39 – Modelando o Algoritmo Alpha-Beta.

38). Além disso, as transições t_3, t_4 e t_5 modelam os três comandos pertencentes ao escopo do comando condicional supra-citado (linhas 3 - 5 da Figura 38).

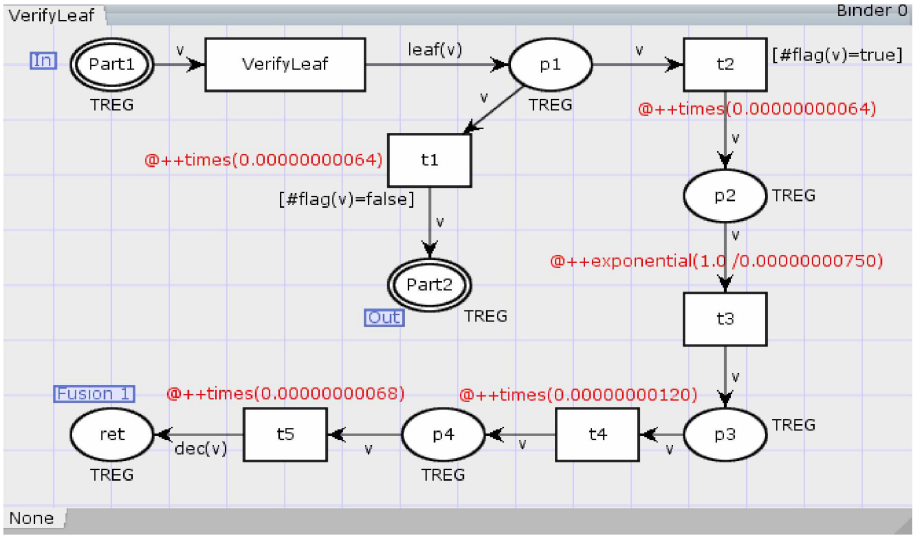


Figura 40 – Modelagem dos Tempos de Execução Reais dos Comandos.

É importante observar que todas as transições que modelam um comando do algoritmo possuem um operador de tempo real específico. Por exemplo, as transições t_1 e t_2 têm uma duração de tempo de $0.64 * 10^{-9}$ segundos cada, associada à função *times*(). Por outro lado, a transição t_3 tem uma duração de tempo de $7.50 * 10^{-9}$ segundos associada à função *exponential*(). Para obter os valores dos tempos de execução de cada comando e considerá-los no modelo, executou-se cinquenta vezes o código C++ correspondente ao algoritmo, de modo que, ao final das execuções, foram calculados os valores médios de cada um desses tempos. Uma vez que a validação desta etapa consiste na comparação do

tempo total de execução do algoritmo com o valor obtido pela simulação do modelo, os custos temporais de cada transição não poderiam ser valores unitários como ocorreu na primeira etapa desta tese (capítulo 5), pois isso não refletiria de forma correta os tempos de execução reais de cada comando do algoritmo.

Todas as transições que modelam um comando tiveram seus operadores de tempos reais associados à função *times*(), exceto a transição t_3 da sub-rede $VerifyLeaf$ da Figura 40, que teve seu operador de tempo real associado à função *exponential*(). Tal estratégia se deve ao seguinte fato: observou-se que, com as cinquenta execuções do algoritmo, todos os comandos gastaram tempos que variavam pouco em torno da média obtida, salvo o comando da linha 3 da Figura 38 (referente à transição t_3). Assim sendo, a função *times*() foi declarada para receber como parâmetro essas médias de tempo e usá-lo em uma distribuição uniforme com intervalo definido por ∓ 0.1 desses valores. Por exemplo, na transição t_5 da Figura 40, a qual modela o comando de retorno recursivo da linha 5 da Figura 38, o operador de tempo real *@++..* [23] foi associado à função *times(0.00000000068)*, cujo parâmetro mostra que o tempo médio calculado para o referido comando ao final das execuções do algoritmo, foi de 0.68 nanosegundos.

Por outro lado, o valor do tempo de execução real do comando da linha 3 da Figura 38 foi obtido por meio do uso da distribuição estocástica produzida pela função *exponential*(). Isso foi necessário porque o referido comando é uma chamada à função *evaluate*() que tem o propósito de avaliar o nó-folha atual, usando uma heurística baseada na execução de uma rede neural artificial específica. Com a realização das cinquenta execuções do algoritmo Alpha-Beta, observou-se que os tempos de execução gastos nas chamadas da função *evaluate*() variaram consideravelmente, não indicando nenhuma proximidade com o comportamento de uma distribuição uniforme. Portanto, realizou-se testes não-paramétricos [43] em algumas distribuições estocásticas para ver qual delas teria uma aderência aceitável para o uso no modelo. Por intermédio do *teste de Kolmogorov-Smirnov*, observou-se que os valores dos tempos de execução nas cinquenta simulações tiveram aderência (com significância $\alpha = 0.01$) a uma distribuição exponencial com valor médio de $7.50 * 10^{-9}$ segundos. Então, um operador de tempo real *@++* foi usado com a função nativa do CPN Tools dada por *exponential*(1.0/0.00000000750), como mostrado na transição t_3 da Figura 40.

Observa-se que além da possibilidade de usar operadores de tempo com valores inteiros, como foi feito na primeira etapa desta tese, o ambiente CPN Tools permite que se faça uso de operadores de tempo com valores reais e estocásticos, como foi o caso da associação do operador de tempo *@++* com as funções de distribuição *times*() e *exponential*() para gerar um parâmetro de tempo com comportamento o mais próximo possível do analisado por meio da execução do algoritmo. Tal proposição não foi sugerida em nenhuma outra proposta citada nos trabalhos correlatos do capítulo 3, configurando uma contribuição nova desta tese.

5.1.2 Modelagem do Fator de Ramificação

Na Figura 38, a terceira e a quarta partes funcionais do algoritmo Alpha-Beta apresentam um comando iterativo cujo fluxo de controle depende, da mesma forma que no algoritmo Minimax tratado na etapa anterior, do fator de ramificação do nó atual. A função de controle $verifLoop(v)$ foi declarada para fazer a verificação da existência ou não de nós filhos do nó atual a serem processados. Porém, diferentemente da primeira etapa, nesta etapa o objetivo é fazer com que o modelo gere em cada simulação, um valor de tempo de execução o mais próximo possível do tempo de execução real do algoritmo, necessitando assim refinar ao máximo os parâmetros do modelo. Com isso, o fator de ramificação do nó atual é obtido no modelo por meio de uma função de controle chamada $sons(v)$ que trabalha também de forma estocástica. Essa necessidade é justificada pelo fato de que a estrutura da árvore explorada pela execução real do algoritmo deve ser muito próxima da estrutura da árvore explorada pela simulação do modelo. Para isso, a função $sons(v)$ usa o método conhecido como *Roulette Wheel* [44], que consiste em gerar, via distribuição uniforme, um número aleatório entre 0 e 1, checando a qual intervalo de probabilidades esse número pertence. Os valores que definem os intervalos de probabilidade foram extraídos das diversas execuções do algoritmo implementado. A Figura 41 mostra parte da estrutura da função de controle $sons(v)$. O primeiro comando condicional checa se o nó atual representado por v é um nó-raiz (que é, $n(v) = 1$) e se o número gerado aleatoriamente pertence ao intervalo de probabilidades $0.0 \leq x < 0.09524$. Se a resposta for afirmativa, a função assume o valor 1 para a variável tf, que representa o número total de nós filhos do nó atual v. Caso contrário, o segundo comando condicional checa se $n(v) = 1$ e se o número aleatório gerado pertence ao intervalo de probabilidades $0.09524 \leq x < 0.14286$. Se a resposta for afirmativa, a função assume o valor 5 para a variável tf. Caso contrário, os comandos condicionais restantes são executados, um a um, até que um valor apropriado seja atribuído à tf.

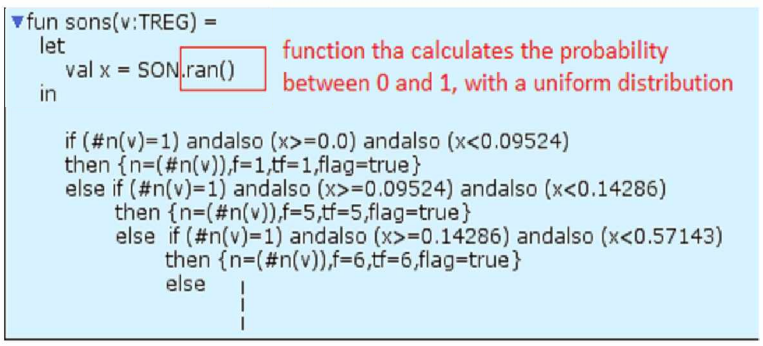


Figura 41 – Estrutura Parcial da Função de Controle sons(v).

É importante destacar que as execuções e simulações feitas neta etapa, consideraram árvores com três níveis fixos, portanto, a geração dos fatores de ramificação se dá apenas

no nó-raiz ($n(v) = 1$) ou nos nós que estão no nível $n(v) = 2$. Os nós que estão no nível $n(v) = 3$ são os nós-folha que, por sua vez, não possuem nós filhos.

5.1.3 Modelagem da Ocorrência de Podas

No algoritmo Alpha-Beta, a ocorrência de podas depende diretamente da ordem dos nós-folha avaliados durante a busca. Nesta tese considerou-se árvores com ordenação aleatória para serem exploradas pelo algoritmo Alpha-Beta. Para representar a exploração de tais árvores no modelo, foram realizadas diversas execuções do algoritmo implementado de modo a extrair as probabilidades associadas com cada tipo de nó produzido. Vale destacar que, quando o nó é do tipo 1 significa que ele possui um nó-filho, quando o nó é do tipo 2 significa que ele possui dois nós-filho e assim por diante.

Na Figura 38, a linha 18 mostra um comando condicional que testa se ocorrerá poda ou não no nó atual. No modelo, esse teste é feito pela função de controle chamada *pruning*(v), mostrada na Figura 42. Se um nó tem um fator de ramificação igual a 4, a função de controle *pruning*(v) chama outra função nomeada por *prob*4(), a qual decidirá se ocorrerá poda ou não no nó atual, baseada na probabilidade associada com esse nó.

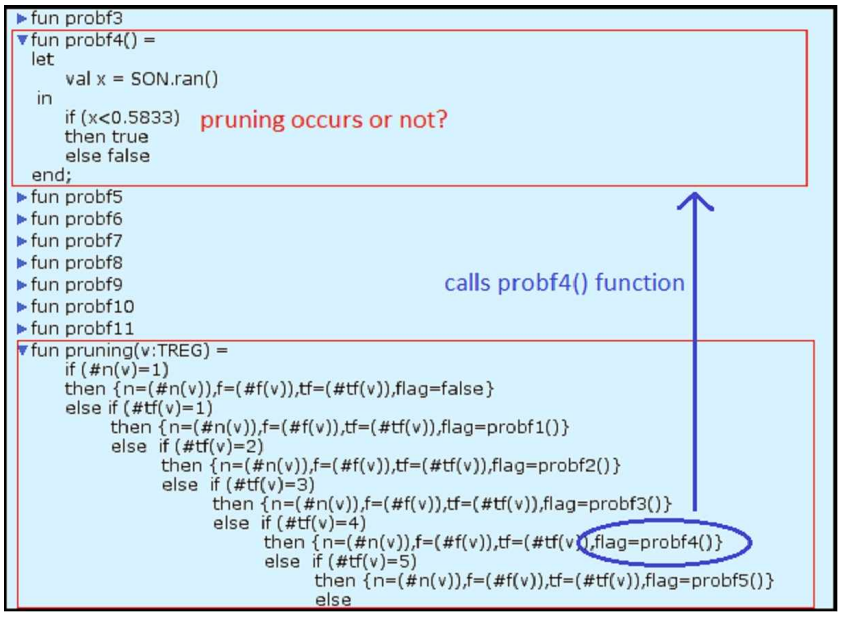


Figura 42 – Estrutura Parcial da Função de Controle pruning(v).

A possibilidade de utilizar no modelo de RdPCH as funções de distribuição presentes na linguagem *Standard ML* e disponíveis no CPN Tools justificaram ainda mais a escolha feita nesta tese para o modelo de RdP e para o ambiente de simulação.

5.2 Simulações do Modelo: Avaliando o Tempo de Execução Real do Alpha-Beta

Finalizada a modelagem do algoritmo Alpha-Beta, considerando sua estrutura, políticas de controle e restrições de tempo, o próximo passo foi a validação da proposta feita nesta etapa. Isso foi realizado por meio da comparação entre os tempos de execução obtidos pelas simulações automáticas do modelo e os tempos de execução reais obtidos nas execuções do algoritmo implementado. Como mostrado na tabela 2, cinquenta execuções foram realizadas em ambos os casos. O tempo de execução médio obtido foi de 0.108962 microssegundos e o obtido das simulações do modelo foi de 0.139189 microssegundos.

Tabela 2 – COMPARATIVO DE TEMPOS DE EXECUÇÃO

Execuções	Tempos de Execução (microsegundos)	
	CPN Tools	Programa
1	0.10356	0.1837
2	0.10544	0.09369
3	0.13846	0.0178
4	0.14695	0.0431
5	0.10398	0.19438
6	0.11252	0.17674
7	0.16445	0.12876
8	0.16801	0.30101
9	0.13846	0.15344
10	0.12148	0.23994
11	0.12232	0.17383
12	0.20207	0.12498
13	0.13044	0.12298
14	0.18118	0.18751
15	0.14836	0.16861
16	0.20301	0.12552
17	0.14836	0.10991
18	0.13002	0.16611
19	0.02017	0.08021
20	0.17665	0.11951
21	0.14779	0.08772
22	0.11252	0.07671
23	0.1296	0.06371
24	0.1743	0.07711
25	0.13898	0.0478
26	0.01975	0.02671
27	0.14799	0.0212
28	0.14554	0.0444
29	0.16534	0.05211
30	0.13898	0.07591
31	0.17509	0.06601
32	0.15591	0.06809
33	0.13804	0.03141
34	0.08705	0.0452
35	0.17561	0.09581
36	0.18363	0.1355
37	0.11257	0.2179
38	0.13232	0.09011
39	0.03767	0.12889
40	0.13096	0.16325
41	0.22665	0.16658
42	0.18269	0.1396
43	0.13002	0.1119
44	0.02871	0.06873
45	0.15534	0.04054
46	0.12903	0.08422
47	0.14133	0.04464
48	0.18321	0.07343
49	0.24499	0.11145
50	0.12195	0.07973
Média	**0.139189**	**0.108962**
Mínimo	0.01975	0.0178
Máximo	0.24499	0.30101
Desvio Padrão	0.045541	0.060723
Coef. de Var.	**0.32719**	**0.55728**

Além disso, os valores do coeficiente de variação obtido em ambos os casos ficaram concentrados em intervalos de tempo bem aproximados (32.71% e 55.72%, respectivamente), como mostrado na tabela 2. Considerando o fato de que as simulações envolveram valores de tempo de execução na ordem de microsegundos, não apenas os tempos de execução médios são próximos, mas também todos os tempos de execução calculados se mantiveram aproximados, como mostrado nas curvas da Figura 43.

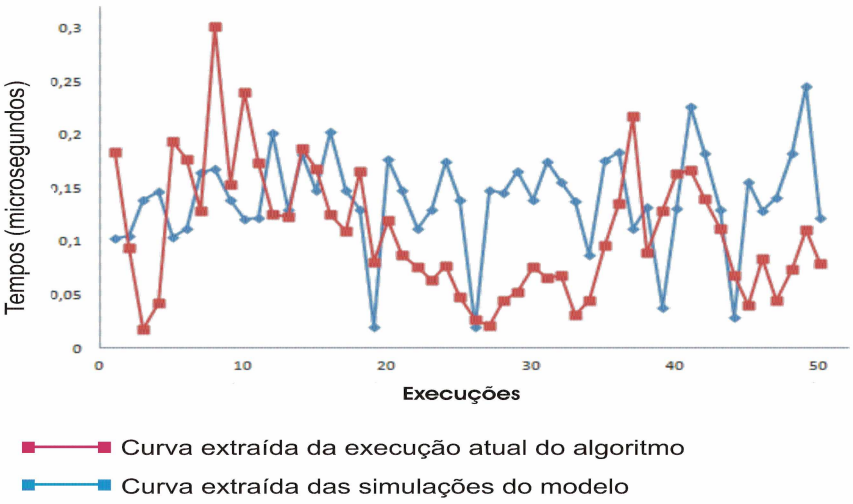


Figura 43 – Curvas dos Tempos de Execução.

5.3 Considerações Relativas ao Capítulo

Este capítulo mostrou como é possível fazer a avaliação do tempo de execução real de um algoritmo por meio de uma análise empírica baseada na simulação automática de modelos formais. Com a realização desta etapa, pôde-se concluir que:

- a RdPCH foi capaz de modelar tanto a estrutura quanto as políticas de controle dos comandos de um algoritmo utilizando inclusive valores de probabilidades e funções de distribuição exigidos na representação eficaz dos desvios intensos nos fluxos de dados durante a sua execução. Isso foi possível graças à possibilidade de se agregar ao modelo as potencialidades existentes na programação funcional *Standard ML* presente no ambiente de simulação CPN Tools. Além disso, foi possível usar operadores de tempo reais para reproduzir de forma aproximada os tempos de execução reais de cada comando do algoritmo modelado; Tais recursos não estão presentes em ferramentas como Autômatos, Grafos ou UML, conforme citado na seção 1.1 desta tese;

- as execuções do algoritmo nesta etapa tiveram o intuito de se extrair valores reais de dados. Utilizando-se tais dados no modelo, os parâmetros de probabilidades, de

funções de distribuição e de operadores de tempo foram bem refinados propiciando uma validação eficaz;

- a corretude da abordagem sugerida nesta etapa foi comprovada, uma vez que as curvas dos tempos de execução reais obtidos com as simulações do modelo e com as execuções do algoritmo implementado mostraram-se muito próximas e, consequentemente, com valores estatísticos bastante aproximados, como foi o caso de ambos valores médios e de ambos coeficientes de variação.

Capítulo 6

Cálculo do Speedup e da Eficiência dos Algoritmos Distribuídos por meio das RdPCH

Este capítulo mostra a terceira etapa de desenvolvimento desta tese, a qual consiste no cálculo do *speedup* e da eficiência dos algoritmos distribuídos por meio de simulações automáticas de modelos de RdPCH, usando o ambiente CPN Tools. O algoritmo usado como estudo de caso, foi uma versão do PVS que apresenta a mesma estrutura de comandos do algoritmo de busca serial Minimax. Com isso, o modelo gerado nesta etapa tem uma estrutura semelhante ao modelo obtido na primeira etapa desta tese, acrescida da modelagem da política de distribuição dos processadores.

Diferentemente da primeira e da segunda etapa, nesta etapa foi necessário incluir no modelo o uso de um recurso disponível no CPN Tools que é a declaração de variáveis globais de modo que a atualização das instâncias das fichas relacionadas ao mecanismo de alocação dos processadores mostrou-se bastante eficaz para a simulação automática dos cenários considerados. Além disso, tais variáveis foram declaradas por meio de estruturas de dados que permitiram representar corretamente o aspecto da sincronização dos processadores existente no algoritmo distribuído PVS.

6.1 Modelagem do Algoritmo PVS

Apesar de existir uma versão do algoritmo PVS que propõe uma distribuição do Minimax com podas (ou seja, do algoritmo Alpha-Beta), esta tese usa como estudo de caso o PVS que distribui o Minimax desconsiderando as podas. A justificativa para tal escolha é de ordem didática: a mera tarefa de modelar e de simular a distribuição de tarefas entre multi-processadores operando em um ambiente de alto desempenho já representa, por si só, um desafio bastante difícil de ser enfrentado. Consequentemente, seria temerário, em um primeiro momento, acrescentar a tal dificuldade o problema das ocorrências das podas

existentes no Alpha-Beta, uma vez que sua solução requer a representação adicional das continuas trocas de mensagem entre os processadores para a atualização de valores *Alpha* e *Beta* da janela de observação.

Nos modelos criados nas duas primeiras etapas desta tese para representar os algoritmos seriais Minimax e Alpha-Beta, todas as declarações foram locais. Porém, para modelar o algoritmo distribuído PVS, é necessário declarar também algumas variáveis globais e novos tipos abstratos de dados que serão apresentados nas próximas seções.

6.1.1 Declarações de Tipos Abstratos de Dados

A fim de permitir que a abordagem proposta lide com a dinâmica de distribuição, foi necessário inserir na estrutura do tipo abstrato REG implementado na seção 4.2 uma variável denominada p. A nova estrutura produzida tem a seguinte forma:

$$REG = \left\{ \; p, \;\; n, \;\; f, \;\; t_f, \;\; flag \; \right\} \tag{5}$$

onde,

- p indica o processador que está sendo usado;
- n indica o nível da árvore que está sendo explorado;
- f indica o nó filho de um nó que está sendo processado;
- t_f indica o total de nós filhos de um nó que está sendo processado;
- $flag$ é usada em várias funções de controle e testada em *Guards* de várias transições.

Uma vez que o modelo deve trabalhar com um operador de tempo, REG deve ser do tipo temporizado e sua estrutura final é então declarada como:

$$TREG = \;\; REG \;\; Timed \tag{6}$$

Outros dois tipos abstratos de dados foram criados: $REG1$ e $REG2$. O tipo $REG1$ foi criado para representar a informação contida em um pilha e é definido como:

$$REG1 = \left\{ \; npar, \;\; fpar, \;\; t_f \; \right\} \tag{7}$$

onde,

- $npar$ indica o nível do PV-node corrente;
- $fpar$ indica o próximo nó filho do PV-node corrente que será distribuído para um processador ocioso;
- t_f indica o total de nós filhos do PV-node corrente.

O tipo $REG2$ foi criado para representar a informação contida em uma lista e é definido como:

$$REG2 = \left\{ \; palloc, \quad npar \; \right\} \tag{8}$$

onde,

- $palloc$ indica qual processador está alocado atualmente;
- $npar$ indica o nível do PV-node corrente.

Na linguagem *Standard ML*, tanto a pilha quanto a lista são declaradas como sendo do tipo $List$. Portanto, o tipo $REG1$ associado à pilha é definido como:

$$LREG1 = \quad List \quad REG1 \tag{9}$$

Da mesma forma, o tipo $REG2$ associado à lista é definido como:

$$LREG2 = \quad List \quad REG2 \tag{10}$$

Apesar de os tipos $LREG1$ e $LREG2$ terem sido declarados como $List$, o primeiro deles, de fato, representa uma pilha, ao passo que o segundo representa uma lista. O quê define essa diferença na funcionalidade entre os dois tipos são as funções de controle que lhes são associadas. De fato, as operações de inserção e de remoção dos dados são específicas para cada caso. Um detalhamento maior sobre o papel da pilha e da lista no modelo é feito posteriormente.

6.1.2 Variáveis Locais

Algumas variáveis locais foram declaradas no modelo, sendo que a principal delas é a variável v. Essa variável é do tipo $TREG$ e está associada a quase todos os arcos do modelo. Seu propósito é carregar as instâncias das fichas quando elas se deslocam de um lugar para outro. Vale lembrar, que $TREG$ é uma extensão temporizada do tipo REG e este último possui todas as informações que as fichas do modelo devem possuir para seus fluxos através do modelo sejam corretos.

6.1.3 Variáveis Globais

Em virtude da característica de sincronização existente no algoritmo PVS, é necessário que algumas variáveis sejam declaradas como globais no modelo. Essas variáveis modelam os efeitos da existência de mensagens entre as partes distribuídas do algoritmo. Portanto, foram criadas quatro variáveis globais:

- np representa a quantidade de processadores secundários ociosos;

- *pvn* é *true* se a exploração na ramificação mais à esquerda da árvore ainda não chegou ao seu nó-folha; caso contrário, é $false$;
- $l1$ representa a pilha que contém os nós filhos de todos os PV-nodes;
- $l2$ representa a lista que contém a relação dos processadores em uso.

Para ilustrar o funcionamento da pilha $l1$, a Figura 44 mostra algumas remoções feitas nessa variável. No exemplo da Figura 8, ao finalizar a exploração de toda a ramificação mais à esquerda da árvore, a pilha $l1$ contém todos os nós a serem distribuídos pelo algoritmo PVS, conforme mostra a Figura 44 (a).

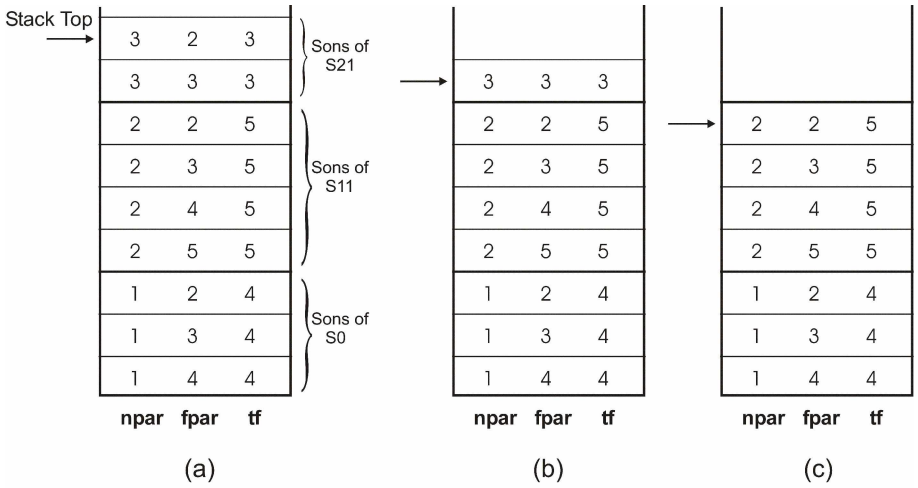


Figura 44 – Pilha de Nós a serem Distribuídos.

Observa-se que, do topo da pilha até a base, a ordem dos nós na pilha indica a ordem com que os nós serão distribuídos, ou seja, sempre priorizando os nós que estão nos níveis mais profundos e mais à esquerda na árvore. De fato, inicialmente, o topo da pilha indica que a distribuição começará com um nó filho do PV-node situado no nível três da árvore ($npar = 3$), ou seja, com um dos filhos do nó s_{21}. Dentre esses nós, é o segundo nó filho mais à esquerda ($fpar = 2$) que será distribuído primeiro para o processador principal $P0$, ou seja, o nó s_{32} (Figura 8). Quando ocorre o primeiro desempilhamento na pilha, o novo topo indica que o próximo nó filho de s_{21} a ser distribuído para $P1$ é o terceiro nó mais à esquerda ($fpar = 3$), conforme mostrado na Figura 44 (b). Finalmente, ao ocorrer o segundo desempilhamento, o topo da pilha mostrada na Figura 44 (c) reflete o fato de que a distribuição passará a ocorrer com os nós filhos do PV-node s_{11} situado no nível dois da árvore ($npar = 2$). Nesse momento, o processador $P0$ passa a explorar o segundo filho mais à esquerda ($fpar = 2$) desse PV-node e o processador $P1$ está ocioso e pronto para processar outro nó.

Durante o processo de distribuição feito pelo PVS, a lista $l2$ indica quais processadores estão alocados para os nós filhos de um PV-node da árvore e em que nível da árvore esse PV-node se encontra. De fato, considerando a primeira distribuição feita na árvore

da Figura 8, a lista $l2$ mostrada na Figura 45 determina que tanto o processador $P0$ ($palloc = 0$) quanto o processador $P1$ ($palloc = 1$) estão alocados para os nós filhos do PV-node situado no nível três da árvore ($npar = 3$).

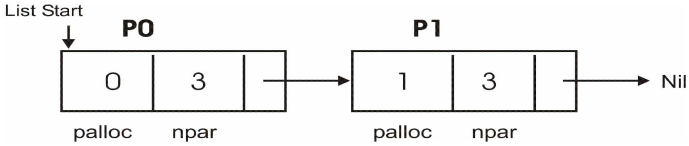


Figura 45 – Lista $l2$ de Processadores em Uso.

A declaração dos tipos abstratos de dados, a criação de funções de controle que realizam as operações de inserção e remoção na pilha $l1$ e na lista $l2$, bem como a declaração de variáveis globais para representar as mensagens existentes entre os processadores, foram fundamentais para a representação da política de distribuição do algoritmo distribuído. O fato de essa potencialidade existente na linguagem *Standard ML* poder ser associada ao modelo de RdPCH por meio do uso do ambiente gráfico CPN Tools foi um dos diferenciais para a escolha de tais ferramentas nesta tese . Saliente-se que tais soluções não seriam passíveis de serem obtidas por meio das abordagens propostas nos trabalhos correlatos apresentados no capítulo 3.

6.1.4 Modelagem do Mecanismo de Alocação de Processadores

Para entender a representação do mecanismo de alocação dos processadores feita no modelo, supõe-se no exemplo da árvore da Figura 8 que o estado atual do processamento seja o seguinte: a distribuição exigida para explorar toda sub-árvore definida a partir do nó s_{11} já foi feita e o próximo passo é distribuir os nós filhos do PV-node s_0 (nós destacados com sombreamento na Figura 8). A partir desse estado do algoritmo, o processador $P0$ é alocado para o processamento do nó s_{12} e, paralelamente, o processador $P1$ é alocado para o processamento do nó s_{13}, conforme mostrado na Figura 8.

Neste estado da simulação, o processador $P0$ é representado por uma ficha no lugar p_3, como mostrado na Figura 46. Devido à extrema complexidade do modelo hierárquico (presença de diversas páginas no modelo), é mostrada apenas uma parte do modelo considerando um certo estado da evolução do algoritmo. Lembrando que o PV-node s_0 é um nó maximizador, a Figura 46 destaca a parte da sub-rede $VerifyMax$ que trata das alocações feitas nos níveis de maximização. Nota-se que a função $genTokens(v)'v$ gerou três fichas no lugar *sons_1* (e, consequentemente, três fichas foram produzidas no lugar *sons_2* que possui a fusão *Fusion2* com o lugar *sons_1*). Os valores $f = 2$ (segundo nó filho), $f = 3$ (terceiro nó filho) e $f = 4$ (quarto nó filho) dessas fichas mostram que elas representam, respectivamente, os nós s_{12}, s_{13} e s_{14} (Figura 8) a serem distribuídos no momento. A Figura 46 representa o momento imediatamente anterior ao processo de distribuição supracitado, pois a transição t_3 está pronta para ser disparada. Das três

fichas situadas em *sons_2*, a ficha que possui $f = 2$ é que será usada nesse disparo da transição t_3. Isso ocorre porque o disparo da transição t_3 só acontece quando as instâncias dos atributos das fichas situadas em seus dois lugares de entrada (p_3 e $sons_2$) coincidem totalmente. Outro ponto importante é que no lugar *alloc_P* (lugar que indica os processadores que estão ociosos), há uma ficha representando o processador $P1$ em estado ocioso. Uma vez que o processador $P0$ nunca está ocioso durante o processamento do algoritmo PVS, a ficha associada ao processador $P0$ nunca estará presente no lugar *alloc_P*.

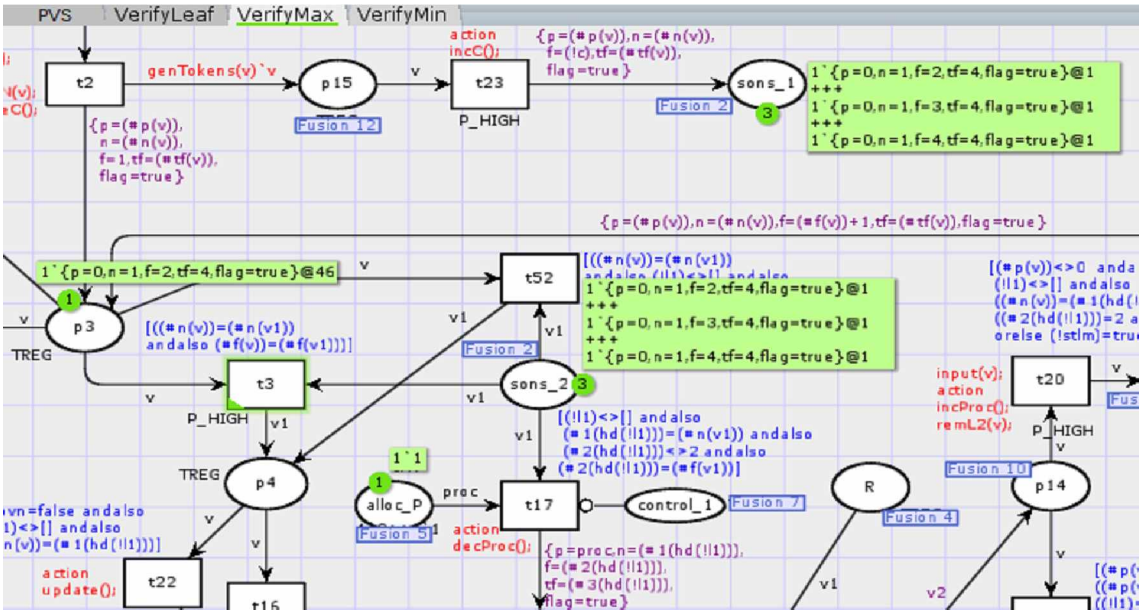


Figura 46 – Estado Imediatamente Anterior à Alocação do Processador P0

A Figura 47 mostra a alocação do processador $P0$ por meio do seguinte processo: ao ocorrer o disparo da transição t_3, uma ficha é produzida no lugar p_4 e os valores dos atributos associados a essa ficha são instanciados. Mais especificamente, $p = 0$ indica que o processador em uso é $P0$, $n = 1$ indica que o PV-node considerado neste momentos é s_0 e, finalmente, $f = 2$ indica que o nó filho que está sendo processado é o segundo filho do PV-node.

A partir desse momento, a dinâmica do modelo evolui para simular a segunda alocação mostrada na Figura 48. Observa-se que a transição t_{17} foi disparada e que os valores dos atributos associados à ficha inserida no lugar p_{13} possuem uma instanciação tal que: $p = 1$ indica que o processador em uso é $P1$, $n = 1$ indica que o PV-node considerado neste momento é s_0 e, finalmente, $f = 3$ indica que o nó filho que está sendo processado é terceiro filho do PV-node.

Por fim, é importante notar que, ao ocorrer as alocações dos processadores $P0$ e $P1$, o lugar *sons_2* ficou com apenas uma ficha (Figura 48), o que representa o fato de que o quarto nó filho ($f = 4$) do PV-node s_0 ainda não foi processado, esperando por uma ficha (processador ocioso) no lugar *alloc_P*.

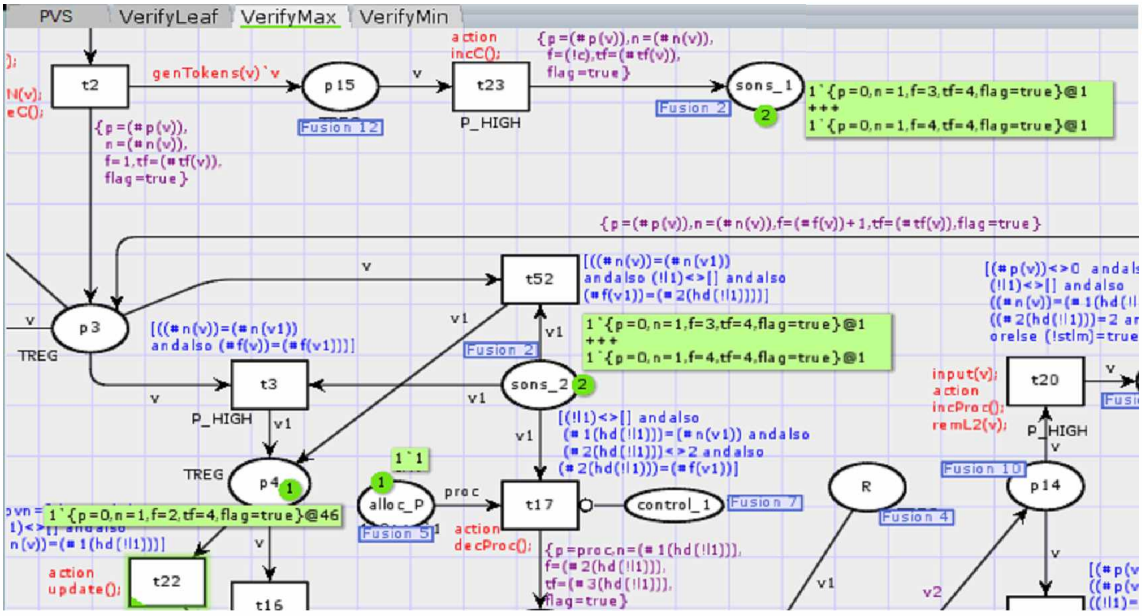


Figura 47 – Alocação do Processador P0

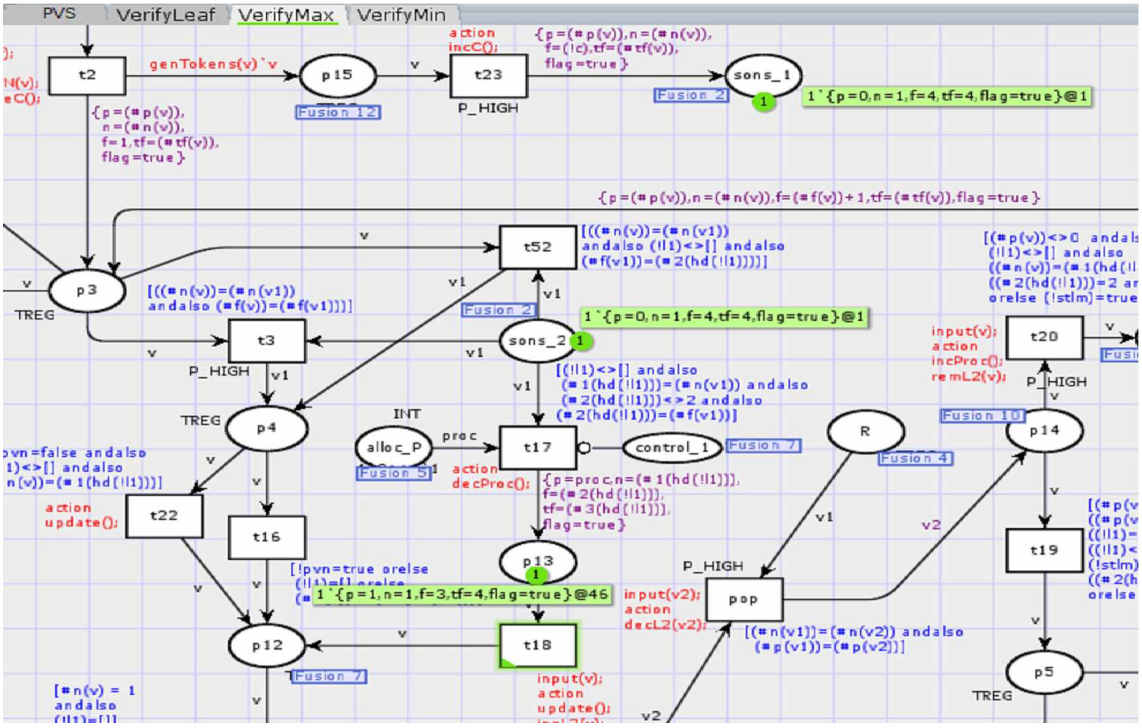


Figura 48 – Alocação do Processador P1

No intuito de aprofundar a compreensão da modelagem da política de distribuição feita neste trabalho, alguns aspectos precisam ser destacados:

- Na Figura 46, as três fichas situadas nos lugares *sons*_1 e *sons*_2 possuem a data $@+1$ e a ficha situada no lugar p_3 possui a data $@+46$. Isso reflete o fato de que as três fichas foram criadas no início da exploração na árvore e que, somente após o processamento de toda subárvore definida a partir do nó s_{11}, é que a terceira

distribuição ocorrerá. Além disso, após o disparo das transições t_3 (alocação de $P0$) e t_{17} (alocação de $P1$), as datas associadas a esses eventos são iguais para ambas as fichas localizadas nos lugares p_4 (Figura 47) e p_{13} (Figura 48), confirmando, assim, a representação do paralelismo existente em ambos os mecanismos de alocação.

- Uma vez que a Figura 48 representa no modelo o estado imediatamente posterior à alocação do processador $P1$, a ficha que estava no lugar p_4, representando o uso do processador $P0$, continuou sua evolução em função da concorrência presente no modelo.

- A Figura 46 mostra a disponibilidade do processador $P1$ no lugar *alloc_P*. Ao ocorrer a liberação do processador $P1$, considerando o efeito apropriado de funções de controle, o processador $P1$ retornará novamente para o lugar *alloc_P* ($P1$ ocioso novamente). Com isso, na sequência, $P1$ será alocado para processar o quarto nó filho do PV-node s_0. O retorno da ficha ao lugar *alloc_P* é originado de outra parte do modelo, uma vez que esse lugar possui uma fusão (*Fusion5*) com outro lugar, como mostrado na Figura 46.

- Além da possibilidade do uso de recursos como operador de tempo, *Guards* e lugares de fusão citados anteriormente, outro recurso fundamental para o modelo proposto nesta etapa são os arcos inibidores que também estão disponíveis no CPN Tools. Na Figura 46, o arco de saída do lugar *control_1* é um arco inibidor. De fato, devido à sobrecarga de sincronização existente no algoritmo PVS, certos disparos de transições precisam ser inibidos até que uma condição aconteça no processamento. Por exemplo, no PVS sempre que um ou mais processadores secundários se tornam ociosos em um nível, não havendo mais nós a serem distribuídos neste nível, tais processadores só poderão processar os nós nos níveis anteriores da árvore após a liberação de uma permissão por parte do processador principal $P0$. Isso foi representado com o uso de arcos inibidores existentes nas RdP e que não estão presentes em ferramentas como UML, autômatos ou grafos.

6.2 Resultados das Simulações

Para validar os resultados experimentais calculados nas simulações, o modelo é simulado em diversos cenários distintos que correspondem a algumas das possíveis execuções do algoritmo para um conjunto representativo de dados.

6.2.1 Simulação com um Processador

Para o primeiro cenário, a exploração das árvores de busca foi simulada com um fator de ramificação médio igual a dois, profundidade máxima variando entre os níveis perten-

centes ao conjunto $\{3, 5, 7, 9, 11, 13, 15\}$ e o uso de apenas um processador. Tal cenário, corresponde à simulação da versão serial do algoritmo PVS (ausência da concorrência devido ao uso de um único processador), o que, na prática, equivale ao algoritmo Minimax sem podas. No intuito de obter resultados consistentes, 30 replicações da simulação foram realizadas para cada profundidade máxima da árvore de busca, de modo a obter os valores médios dos tempos de execução produzidos.

Conforme esperado, esse cenário gerou os mesmos resultados produzidos na primeira etapa desta tese, na qual a curva da função de complexidade do tempo de execução obtida nas simulações do modelo mostrou-se muito próxima da curva obtida na análise teórica (seção 4.3).

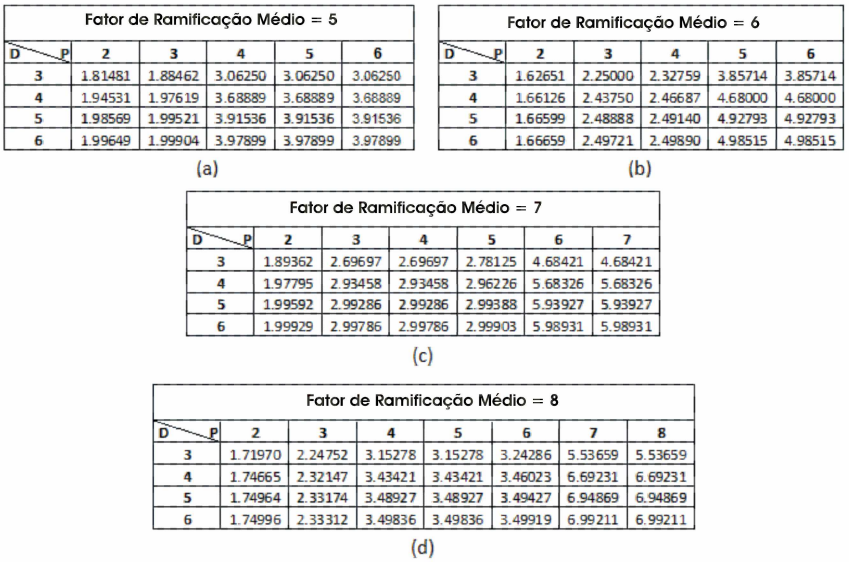
6.2.2 Simulação com Diversos Processadores

Os demais cenários simulados consideraram as políticas de distribuição sugerida pelo algoritmo PVS com o uso de mais de um processador. As tabelas da Figura 49 (a), (b), (c) e (d) mostram, respectivamente, os resultados obtidos na simulação de quatro cenários distintos. Para cada cenário, as árvores de buscas foram exploradas com um fator de ramificação médio específico calculado durante a simulação pela função de controle $defsons(v)$ citada na seção 4.2.2. Uma vez que o modelo representa processos concorrentes, o fator de ramificação médio calculado precisa ser representado fisicamente por fichas, o que é feito pela função de controle $genTokens(v)'s$ citada na seção 6.1.2. Por exemplo, se a função $defsons(s)$ calcula um fator de ramificação médio igual a 5, de posse desse valor a função $genTokens(v)'v$ produzirá 5 fichas distintas (cada uma representando um nó filho do nó atual).

Analisando os resultados mostrados na tabela da Figura 49 (a), nota-se que os valores de *speedup* tendem a aumentar quando há um aumento no número de processadores P. O valor de *speedup* máximo é obtido quando P é igual ao valor do fator de ramificação médio decrementado de uma unidade, ou seja, quando $P = 4$ processadores. A partir daí, o aumento de P não produziu o aumento do *speedup*, indicando a saturação no ganho de desempenho do algoritmo PVS. Isso confirma a característica desse algoritmo citada na seção 2.5 que é a diminuição de sua eficiência quando P torna-se maior ou igual ao fator de ramificação médio da árvore.

A Figura 50, apresenta as curvas relacionadas às eficiências produzidas pelo PVS em cada um dos quatro cenários. Avaliando essas curvas conclui-se que:

- Quanto mais profunda for a busca na árvore, maior é a eficiência do PVS;
- O PVS apresenta uma eficiência mínima quando P é igual ao fator de ramificação médio decrementado de duas unidades. Por exemplo, o cenário mostrado na tabela da Figura 50 (d) cujas árvores possuem um fator de ramificação médio igual a oito, indica que quando $P = 6$ processadores a eficiência produzida é mínima. Isso



Fator de Ramificação Médio = 5

D \ P	2	3	4	5	6
3	1.81481	1.88462	3.06250	3.06250	3.06250
4	1.94531	1.97619	3.68889	3.68889	3.68889
5	1.98569	1.99521	3.91536	3.91536	3.91536
6	1.99649	1.99904	3.97899	3.97899	3.97899

(a)

Fator de Ramificação Médio = 6

D \ P	2	3	4	5	6
3	1.62651	2.25000	2.32759	3.85714	3.85714
4	1.66126	2.43750	2.46687	4.68000	4.68000
5	1.66599	2.48888	2.49140	4.92793	4.92793
6	1.66659	2.49721	2.49890	4.98515	4.98515

(b)

Fator de Ramificação Médio = 7

D \ P	2	3	4	5	6	7
3	1.89362	2.69697	2.69697	2.78125	4.68421	4.68421
4	1.97795	2.93458	2.93458	2.96226	5.68326	5.68326
5	1.99592	2.99286	2.99286	2.99388	5.93927	5.93927
6	1.99929	2.99786	2.99786	2.99903	5.98931	5.98931

(c)

Fator de Ramificação Médio = 8

D \ P	2	3	4	5	6	7	8
3	1.71970	2.24752	3.15278	3.15278	3.24286	5.53659	5.53659
4	1.74665	2.32147	3.43421	3.43421	3.46023	6.69231	6.69231
5	1.74964	2.33174	3.48927	3.48927	3.49427	6.94869	6.94869
6	1.74996	2.33312	3.49836	3.49836	3.49919	6.99211	6.99211

(d)

Figura 49 – Speedup do Algoritmo PVS.

ocorre porque, durante o processamento do nó filho mais à esquerda de um PV-node, restam sete nós filhos para serem distribuídos. Quando os seis processadores disponíveis são distribuídos, o nó filho remanescente não será processado até que um desses processadores torne-se ocioso. Quando um processador explorar o nó filho remanescente, os outros cinco processadores tendem a ficar ociosos até o fim dessa exploração. Essa é a principal causa para o decréscimo da eficiência do algoritmo PVS;

- ❑ A melhor eficiência ocorre em árvores com um fator de ramificação médio ímpar. Tal eficiência é maximizada quando o número de processadores P é igual a um valor divisor do fator de ramificação médio decrementado de uma unidade. Por exemplo, na Figura 50 (c) em que o fator de ramificação médio é igual a sete, para $P = 2$, $P = 3$ e $P = 6$ processadores, a eficiência obtida é máxima.

Além dessas características, um aspecto muito importante que reforça a validação da modelagem da política de distribuição do PVS, é a detecção do *overhead* de sincronização, citado na seção 2.5. A Figura 51, mostra as curvas do *speedup* e da eficiência obtidos em simulações que explora uma árvore com fator de ramificação médio igual a treze e profundidade máxima igual a seis. Nota-se que quando P aumenta e os valores de *speedup* permanecem constantes, isso faz com que a eficiência do PVS diminua. De fato, para $4 \leq P \leq 5$, $6 \leq P \leq 11$ e $P \geq 12$, a curva da Figura 51 (a) mostra que os *speedup* são constantes, fazendo com que a eficiência do PVS diminua (Figura 51 (b)). Isso reflete situações em que existem processadores ociosos esperando um evento ocorrer para que eles sejam efetivamente usados no processamento, evidenciando o *overhead* de sincronização mencionado.

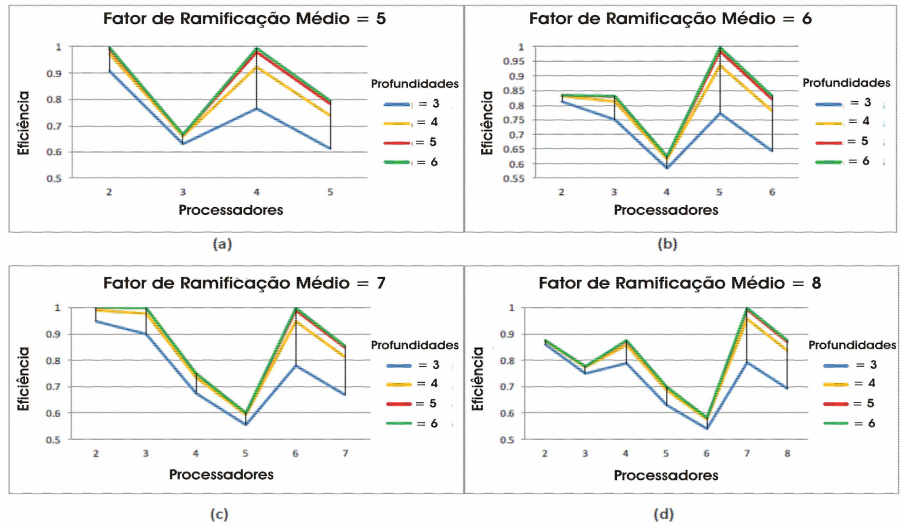


Figura 50 – Eficiência do Algoritmo PVS.

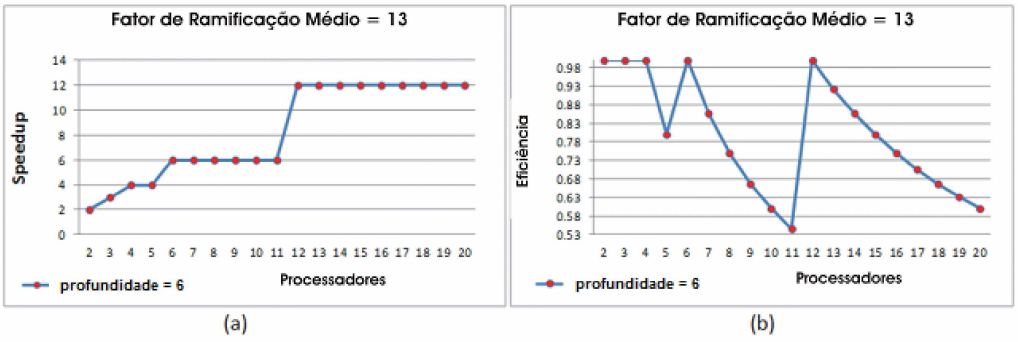


Figura 51 – Comparativo das Curvas: (a) Speedup (b) Eficiência.

6.3 Considerações Relativas ao Capítulo

Este capítulo mostrou como é possível fazer o cálculo do *speedup* e da eficiência de um algoritmo distribuído por meio de uma análise empírica baseada na simulação automática de modelos formais. Com a realização desta etapa, concluiu-se que:

- a RdPCH foi capaz de modelar a política de distribuição do algoritmo PVS, de modo que as simulações automáticas desse modelo refletiram de forma bem aproximada o comportamento real da execução do algoritmo PVS;

- o uso de variáveis globais proporcionou a representação da comunicação exigida na distribuição dos processadores, gerando o sincronismo existente no algoritmo. A possibilidade de representar a alocação de recursos, o paralelismo e o sincronismo por meio de um modelo de RdP, assim como de declarar variáveis globais usando a linguagem *Standard ML* e de usar arcos inibidores presentes nas RdPCH, foram fundamentais para a criação do modelo cuja dinâmica reproduziu de modo correto o funcionamento da política de distribuição do algoritmo distribuído. Esses recursos

não estão presentes em ferramentas usadas nos trabalhos correlatos citados na seção 3;

- a análise do comportamento do algoritmo PVS não exigiu a sua implementação, pois, por meio das simulações realizadas no modelo, foi possível verificar as suas principais características;
- os valores dos *speedup* e das eficiências, obtidos por meio da análise empírica proposta nesta tese, confirmaram a principal característica do algoritmo PVS que é a sobrecarga de sincronismo na situação extrema de ausência de podas (comportamento do Minimax).

Capítulo 7

Conclusões e Trabalhos Futuros

Este trabalho apresentou uma abordagem baseada em simulações de modelos de RdPCH que permite a avaliação do desempenho de algoritmos seriais e distribuídos, sem a necessidade de implementá-los. Com a criação dos modelos foi possível reproduzir de forma eficaz, por meio de simulações automáticas de diversos cenários completos, os tempos de execução dos algoritmos usados como estudo de casos. Foram considerados algoritmos com complexas estruturas de comandos processados de forma serial e distribuída.

O uso das RdPCH permitiu uma representação estrutural mais compacta e intuitiva facilitando a criação e a adequação dos mesmos. Para reproduzir os diversos cenários, foi escolhido o ambiente CPN Tools que mostrou-se muito eficaz por disponibilizar diversos recursos gráficos responsáveis pela apresentação visual dos modelos e recursos de simulação que permitiram a obtenção dos tempos de execução.

7.1 Considerações Finais

Com a realização da primeira etapa deste trabalho apresentada no capítulo 4, as RdPCH permitiram modelar corretamente a estrutura dos comandos e dos fluxos de controles do algoritmo de busca serial Minimax. Além disso, o uso de recursos como cores, *Guards*, inscrições em arcos, lugares de fusão, transições abstratas, *sockets* e declarações de tipos de dados, funções de controle e variáveis locais, viabilizou no modelo o acompanhamento da dinâmica desse algoritmo. Para calcular o tempo de execução foram necessárias diversas simulações automáticas do modelo através do ambiente CPN Tools. Tal ambiente permitiu a associação de um operador de tempo com custo unitário em cada transição que modela um comando do algoritmo. Assim, foi possível contabilizar de forma correta os tempos gastos em cada cenário considerado. Em especial, para tornar mais amplo o escopo de aplicação da abordagem sugerida nesta etapa, os cenários simulados consideraram a exploração de árvores não-uniformes. Isto só foi possível porque o CPN Tools permitiu utilizar nos modelos, determinadas funções de distribuição

estocásticas nativas da linguagem *Standard ML*. A validação desta etapa consistiu na comparação entre a curva da função de complexidade obtida teoricamente na literatura e a curva obtida experimentalmente por meio das simulações do modelo. Tais curvas apresentaram um comportamento muito próximo entre elas confirmando assim, a corretude do método proposto. Vale destacar que nesta etapa não houve a necessidade de implementar o algoritmo modelado.

Considerando os resultados mostrados no capítulo 5 referente à segunda etapa deste trabalho, utilizou-se como estudo de caso o algoritmo de busca serial Alpha-Beta que possui a mesma complexidade estrutural de comandos do algoritmo Minimax acrescida dos comandos responsáveis para a realização das podas na árvore de busca. Portanto, além dos recursos usados na primeira etapa e que foram fundamentais para modelar o algoritmo Alpha-Beta, em especial, o uso de probabilidades e de funções de distribuição permitiu a representação da ocorrência das podas existentes nele. A validação da abordagem feita nesta etapa consistiu na comparação dos tempos de execução reais do algoritmo em diversos cenários com os tempos de execução extraídos nas simulações dos modelos para os mesmos cenários. Desta forma, foram utilizados nos modelos operadores de tempo reais disponíveis no CPN Tools e estes foram associados a cada transição que modela um comando do algoritmo. As curvas obtidas em ambos os casos mostraram-se estatisticamente próximas.

Por fim, com a realização da terceira etapa citada no capítulo 6 foi possível calcular experimentalmente os valores do *speedup* e da eficiência do algoritmo de busca distribuída PVS. A representação da política de distribuição exigida na modelagem desse algoritmo foi realizada por meio das RdPCH cujas simulações foram realizadas no ambiente CPN Tools. O recurso de declaração de variáveis globais neste ambiente foi fundamental para representar as sincronizações dos processadores considerados nos cenários simulados. Tais simulações, isentas da necessidade da implementação do algoritmo distribuído PVS, produziram resultados que confirmaram as principais características desse algoritmo já sinalizadas em outros trabalhos científicos.

7.2 Contribuições Científicas

A abordagem sugerida nesta tese propiciou duas principais contribuições científicas:

- a avaliação do tempo de execução de algoritmos seriais com estruturas complexas de comandos por meio da simulação automática de modelos considerando inclusive tempos de execução reais;

- o cálculo dos valores do *speedup* e da eficiência dos algoritmos distribuídos por meio da simulação automática de modelos, sem a necessidade de implementá-los.

7.3 Limitações

Uma limitação encontrada no desenvolvimento da primeira e da terceira etapa desta tese, foi a impossibilidade de simular cenários envolvendo situações extremas. Apesar dos experimentos realizados terem considerado árvores de busca com diversos fatores de ramificação médios e diversas profundidades máximas, por questão de limitação do ambiente CPN Tools, não foi possível simular árvores maiores do que as apresentadas nos experimentos. O fato é que a ferramenta de simulação deste ambiente possui um limite para a contagem dos custos temporais envolvidos. Porém, vale destacar que tal limite não inviabilizou a proposta sugerida e permitiu validar todas as abordagens feitas. Ou seja, esta é uma limitação do ambiente e não do modelo formal escolhido.

7.4 Trabalhos Futuros

Durante a realização das etapas desenvolvidas, algumas questões foram levantadas e não foram investigadas por não fazerem parte do escopo previsto nesta pesquisa. Portanto, tais questões configuram propostas de trabalhos a serem realizados no futuro e que são listadas a seguir:

- Modelar o algoritmo distribuído PVS incluindo as podas efetuadas durante a exploração da árvore de busca permitindo comparar os resultados obtidos por meio das simulações automáticas do modelo com os resultados conhecidos na literatura;
- Produzir ambientes gráficos de análise de desempenho de algoritmos que permitam não somente a simulação automática, mas também a criação automática de modelos a partir do carregamento desses algoritmos. Uma vez que já existem especificações para converter um modelo de RdPC no formato PNML, sugere-se usá-lo em conjunto com a semântica sugerida nesta tese para produzir tais ambientes;
- Calcular o tempo de execução real de algoritmos sem a necessidade de implementá-los para a realização da validação da modelagem. Existem modelos de RdPC que possuem extensões capazes de lidarem com durações de disparos de transições na forma estocástica. Há também modelos de RdP que permitem associar às transições, durações de tempo com valores específicos ou com valores dentro de intervalos contínuos como no caso das *Redes de Petri t-Temporizada* ou das *Redes de Petri t-Temporal*, respectivamente. Com base em simulações, pode-se pensar em refinar as durações de disparos das transições de modo a obter tempos de execução do modelo o mais próximos possíveis dos tempos reais de execução do algoritmo;
- Paralelizar algoritmos seriais conhecidos propondo uma nova política de distribuição por meio de seus modelos. Isto pode ser tentado com base na verificação de propriedades do modelo existentes nas RdP, como invariantes de lugar;

- Modelar e avaliar o desempenho de algoritmos distribuídos com políticas de distribuição mais sofisticadas como no caso do algoritmo *Young Brothers Wait Concept* (YBWC) [45] que se baseia na troca intensiva de mensagens ou que ainda não foram implementados por questões de limitações do *hardware* existente, como no caso do algoritmo *Dynamic Tree Splitting* (DTS) [46].

Referências

1 CORMEN, T. H. et al. *Introduction to Algorithms.* [S.l.]: MIT Press, 2009.

2 BARBOSA, V. C. *An Introduction to Distributed Algorithms.* [S.l.]: The MIT Press, 2003.

3 GOLDSMITH, S. F.; AIKEN, A. S.; WILKERSON, D. S. Measuring empirical computational complexity. In: *Proceeding ESEC-FSE07.* [S.l.: s.n.], 2007. p. 395–404.

4 COULOURIS, G. et al. *Distributed Systems: Concepts and Design.* 5th. ed. [S.l.]: Pearson, 2011.

5 BOROVSKA, P.; LAZAROVA, M. Efficiency of parallel minimax algorithm for game tree search. In: *International Conf. on CompSysTech.* New York, USA: [s.n.], 2007. p. 1–6.

6 FERREIRA, A.; MONTEIRO, A. G. e J. On the evaluation of shortest journeys in dynamic networks. *Sixth IEEE International Symposium on Network Computing and Applications*, p. 3–10, 2007.

7 HOPCROFT, J. E.; ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation.* [S.l.]: Menlo Park: Addison-Wesley, 1979.

8 ZHAO, X.; AL et. Research on software behavior modeling based on extended finite state automata. In: *IEEE Communications Security Conference.* [S.l.: s.n.], 2014. p. 1–5.

9 RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. *The Unified Modeling Language Reference Manual.* [S.l.]: Addison-Wesley Prof., 2004.

10 MALLOY, B. A.; POWER, J. F. Exploiting uml dynamic object modeling for the visualization of c++ programs. In: *SoftVis-05 Proceedings of the 2005 ACM symposium on Software visualization.* [S.l.: s.n.], 2005. p. 105–114.

11 RUSSELL, S.; NORVIG, P. [S.l.]: Prentice Hall.

12 SCHAEFER, J.; PLAAT, A. New advances in alpha-beta searching. In: *24th Annual Conference on Computer Science of the ACM.* Pennsylvania, USA: [s.n.], 1996. p. 124–130.

13 MARSLAND, T. A.; CAMPBEL, M. S. Parallel search of strongly ordered game trees. In: *ACM Computing Surveys.* [S.l.: s.n.], 1982. v. 14, p. 533–551.

14 ROCKI, K. M. *Large Scale Monte Carlo Tree Search on GPU.* Dissertação (Mestrado) — Departament of Computer Science, School of Information Science and Technology, The University of Tokyo, Tokyo, Japan, 2011.

15 NETO, H. C.; AL et. Ls-visiondraughts: Improving the performance of an agent for checkers by integrating computational intelligence, reinforcement learning and a powerful search method. *Applied Intelligence (APIN)*, v. 41, n. 2, p. 525–550, 2014.

16 HASAN, K. S.; AL et. Implementation of a distributed chess playing software system using principal variation splitting. In: *16th PDPTA*. Nevada, USA: [s.n.], 2010.

17 D., V. A. R.; OTSUKA, A. B.; JULIA, R. M. S. Improvement obtained by ordering the tree-search of an agent for draughts which learns by reinforcement. In: *In: XXXV Conferência Latino-Americana de Informática (CLEI)*. [S.l.: s.n.], 2009.

18 MURATA, T. Petri nets: Properties, analysis and applications. In: *Proceedings of the IEEE*. [S.l.: s.n.], 1989. v. 77, p. 541–580.

19 KISSOUM, Y.; SAHNOUN, Z. A recursive colored petri nets semantics for auml as base of test case generation. In: *AICCSA*. [S.l.: s.n.], 2008. p. 785–792.

20 WENDT, S. Modified petri nets as flowcharts for recursive programs. *Journal Software - Practive and Experience*, v. 10, p. 935–942, 1980.

21 HANZáLEK, Z. *Parallel Algorithms for Distributed Control - A Petri Net Based Approach.* Tese (Doutorado) — Czech Technical University, Prague, Czech Republic, 2003.

22 ZHU, L.; TONG, W.; CHENG, B. Cpn tools' application in verification of parallel programs. In: *ICICA 2010, Part I, CCIS 105*. [S.l.: s.n.], 2010. p. 137–143.

23 JENSEN, K.; KRISTENSEN, L. M. *Coloured Petri Nets - Modeling and Validation of Concurrent Systems.* [S.l.]: Springer Verlag, 2009.

24 GROUP, A. I. S. *CPN Tools.* 2015. Disponível em: <http://cpntools.org>.

25 LYNCH, M.; GRIFFITH, N. Neurodraughts: the role of representation, search, training regime and architecture in a td draughts player. In: *Eighth Ireland Conference on Artificial Intelligence*. [S.l.: s.n.], 1997. p. 64–72.

26 NETO, H. C.; JULIA, R. M. S. Ls-draughts - a draughts learning system based on genetics algorithms, neural network and temporal differences. In: *IEEE Congress on Evolutionary Computation*. [S.l.: s.n.], 2007. p. 2523–2529.

27 CAEXETA, G. S.; JULIA, R. M. S. A draughts learning system based on neural networks and temporal differences: The impact of an efficiente tree-search algorithm. In: *LNAI Series: Springer Verlag*. New York: [s.n.], 2008. v. 5249, p. 73–82.

28 PLAAT, A. *Research Re: Search and Re-Search.* Dissertação (Mestrado) — Erasmus University, Amsterdan, Holland, 1996.

29 FULLER, S. H.; GASCHNIG, J. G.; GILLOGLY, J. J. Technical Report, *Analysis of the Alpha-Beta Pruning Algorithm.* Pittsburg: [s.n.], 1973.

30 BARCELOS, A. R. A.; JULIA, R. M. S.; JUNIOR, R. M. Dvisiondraughts: a draughts player neural network that learns by reinforcement in a high performance environment. In: *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2011 Proceedings*. Bruges, Belgium: [s.n.], 2011. p. 135–140.

31 BROCKINGTON, M. G. *Asynchronous Parallel Game Tree Search*. Tese (Doutorado) — University of Alberta, Alberta, Canada, 1998.

32 KNUTH, D. E.; MOORE, R. W. An analysis of alpha-beta pruning. In: *Artificial Inteligence*. [S.l.: s.n.], 1975. p. 293–326.

33 MANOHARARAJAH, V. *Parallel Alpha-Beta Search on Search on Shared Memory Multiprocessors*. Dissertação (Mestrado) — University of Toronto, Toronto, Canada, 2001.

34 GUAN, H.; LI, C.-K.; CHEUNG, T.-Y. A generalized stochastic petri net approach for modeling multi-processor parallel processing systems. In: *IEEE 2nd International Conference on Architectures for Parallel Processing*. [S.l.: s.n.], 1996. p. 413–420.

35 SHEN, V.; CHANG, Y.-S.; JUANG, T.-Y. Supervised and unsupervised learning by using petri nets. In: *IEEE Trans. Syst. Man Cybern. A Syst. Humans*. [S.l.: s.n.], 2010. v. 40, n. 2, p. 363–375.

36 JIAN-WEI, Y. et al. A distributed fault diagnosis approach in power system based on fuzzy reasoning petri net. In: *Power and Energy Engineering Conference (APPEEC)*. [S.l.: s.n.], 2010. p. 1–4.

37 SEDDR, I. J. *Proposal for the Subdivision of Project 7.19 for a Petri Net Standard*. 2017. Disponível em: <http://www.petrinets.info/docs/proposal.html>.

38 TGI-GROUP. *Petri Nets - Standardization*. 2017. Disponível em: <http://www.petrinets.info/standard.php>.

39 AALST, W. v. d.; STAHI, C. *Modeling Business Processes: A Petri Net-Oriented Approach*. [S.l.]: The MIT Press, 2011.

40 JENSEN, K. *An Introduction to the Theoretical Aspects of Coloured Petri Nets*. 2017. Disponível em: <http://www.daimi.au.dk/~kjensen/papers_books/rec_papers_books.html>.

41 JENSEN, K.; KRISTENSEN, L. M.; WELLS, L. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. p. 213–254, 2007.

42 GROUP, A. I. S. *An Extension of Standard-ml*. 2017. Disponível em: <http://cpntools.org/trash/an_extension_of_standard_ml>.

43 SPRENT, P.; SMEETON, N. C. *Applied Nonparametric Statistical Methods*. [S.l.]: Chapman and Hall/CRC, 2007.

44 GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. [S.l.]: Addison Wesley Longman, Inc., 1989.

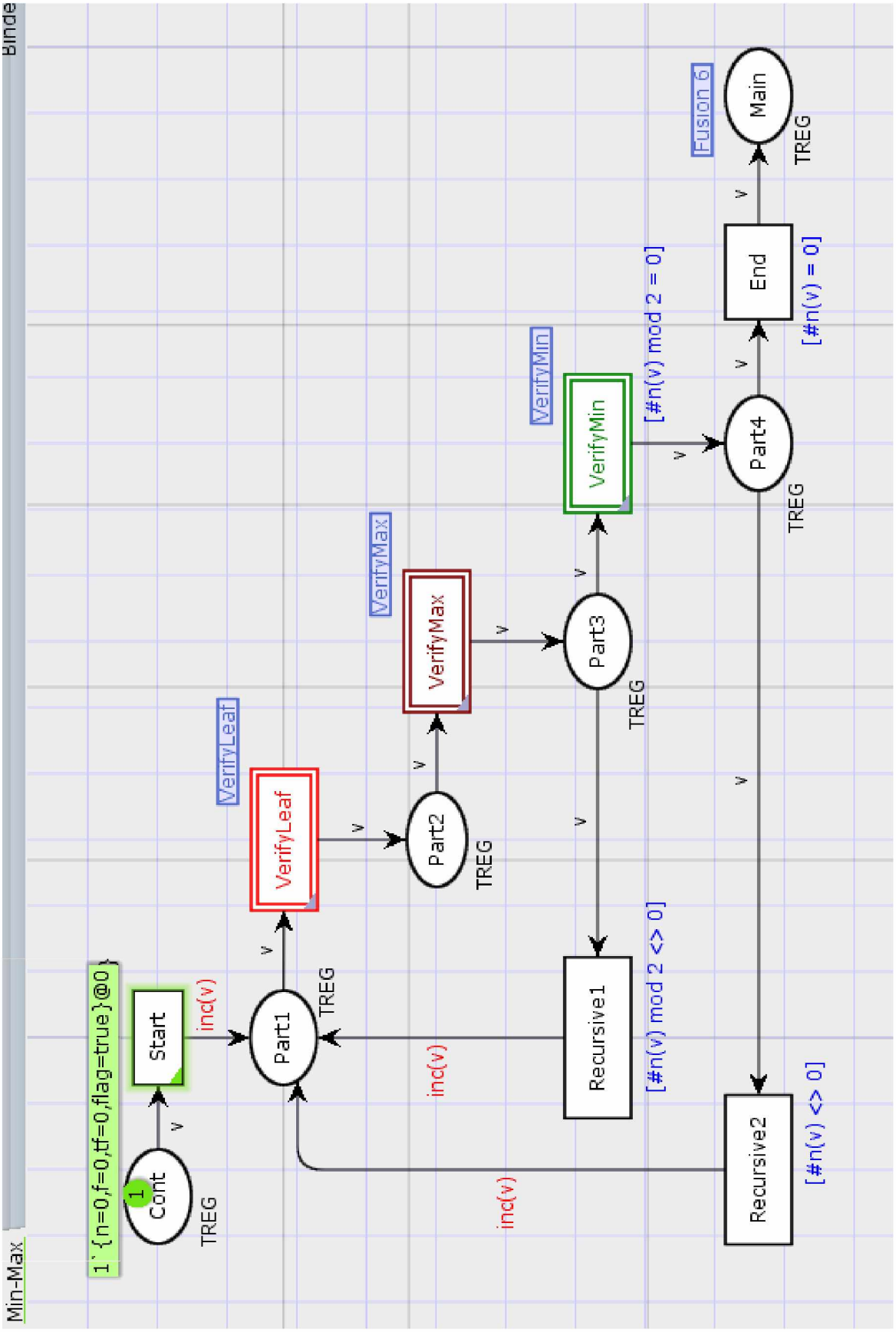
45 FELDMANN, R. et al. Distributed game-tree search. *ICCA, 12*, p. 65–73, 1989.

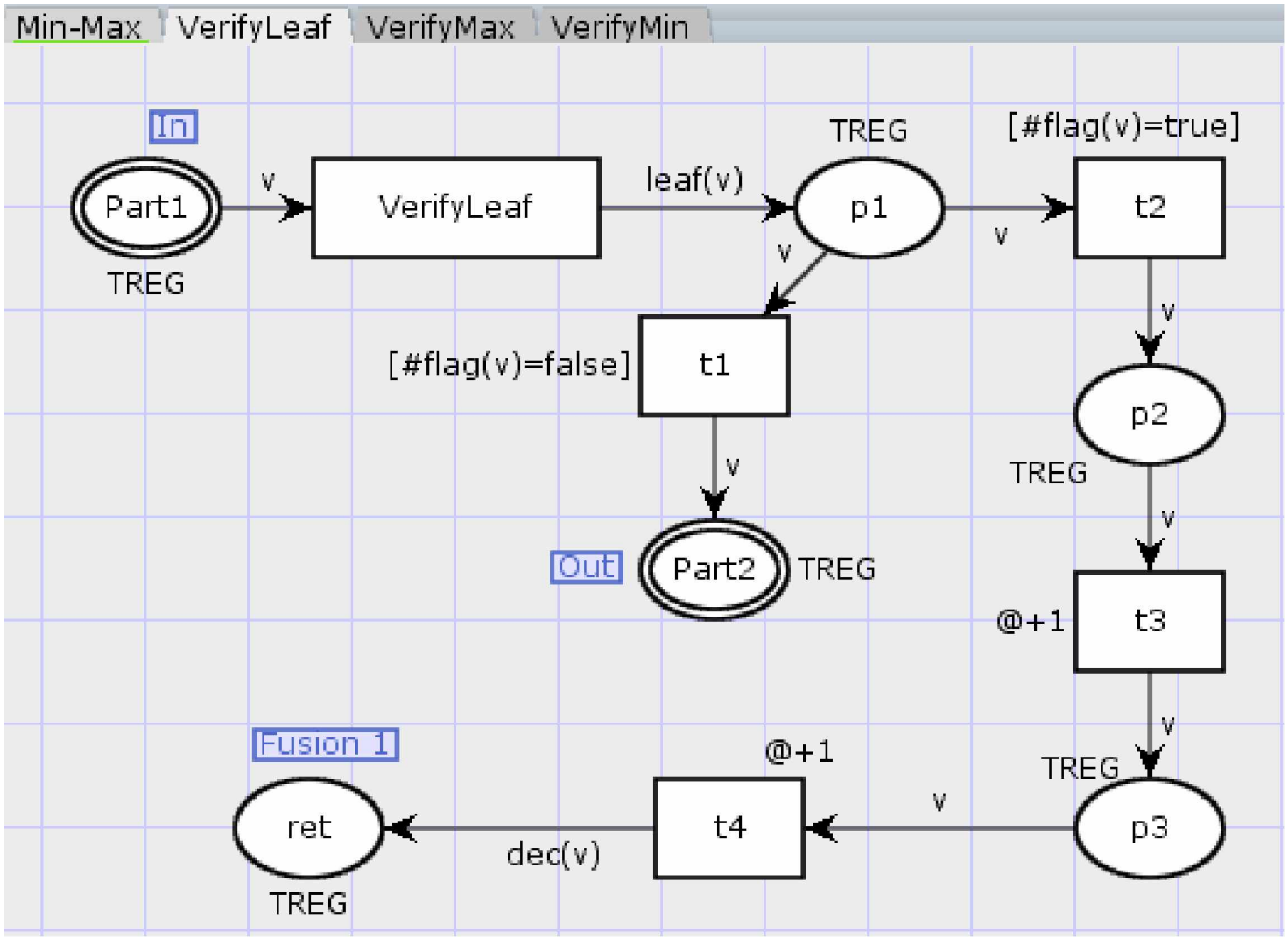
46 HYATT, R. M. *A High-Performance Parallel Algorithm to Search Depth-First Game Trees*. Tese (Doutorado) — University of Alabama, Birmingham, USA, 1988.

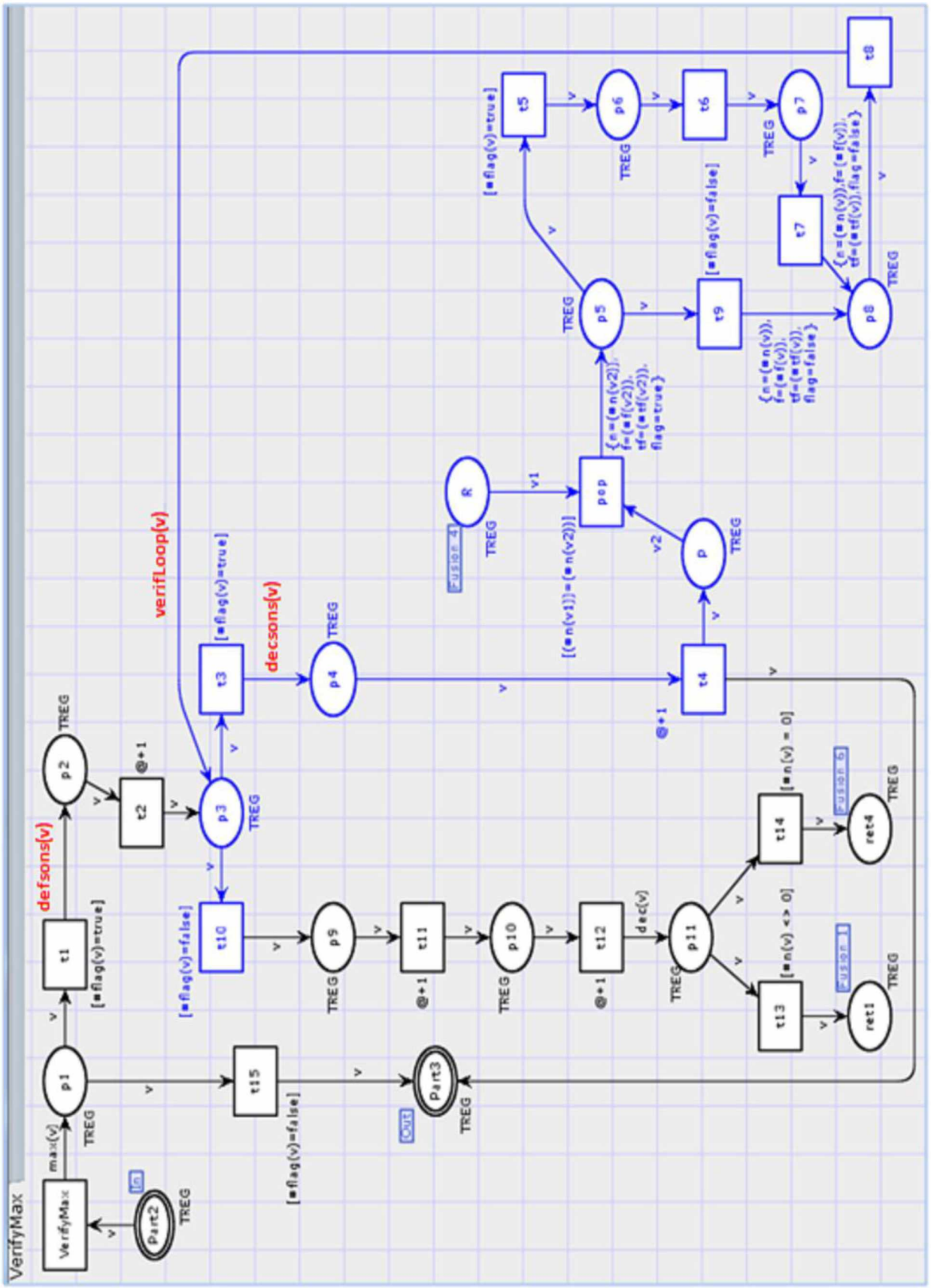
Anexos

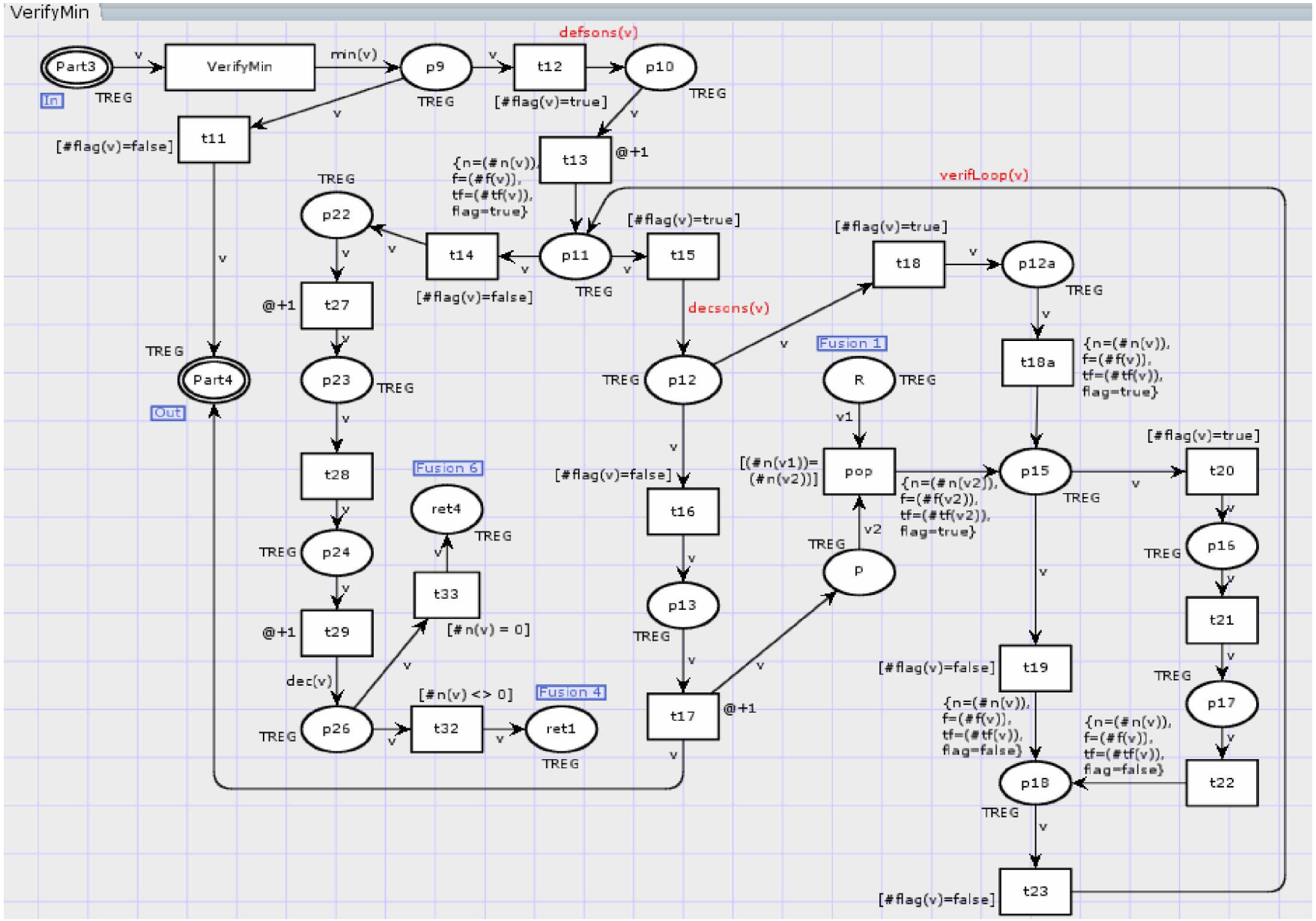
ANEXO **A**

Modelagem Completa da RdPCH Referente ao Minimax


Min-Max
1`{n=0,f=0,tf=0,flag=true}@0
Cont
TREG
Start
inc(v)
Part1
TREG
VerifyLeaf
VerifyLeaf
Part2
TREG
VerifyMax
VerifyMax
Part3
TREG
VerifyMin
VerifyMin
Part4
TREG
End
Main
TREG
Fusion 6
Recursive1
Recursive2
[#n(v) mod 2 <> 0]
[#n(v) mod 2 = 0]
[#n(v) <> 0]
[#n(v) = 0]
inc(v)
inc(v)
v


Min-Max
VerifyLeaf
VerifyMax
VerifyMin
In
Part1
TREG
v
VerifyLeaf
leaf(v)
TREG
p1
v
[#flag(v)=true]
t2
v
v
p2
TREG
v
[#flag(v)=false]
t1
v
Out
Part2
TREG
@+1
t3
v
Fusion 1
ret
TREG
dec(v)
@+1
t4
v
TREG
p3


VerifyMax
VerifyMax
max(v)
p1
TREG
t1
defsons(v)
[•flag(v)=true]
p2
TREG
t2
@+1
p3
TREG
verifLoop(v)
t3
[•flag(v)=true]
decsons(v)
p4
TREG
t4
@+1
p
TREG
v2
pop
v1
R
TREG
Fusion 4
[((•n(v1))=(•n(v2)))]
{n=(•n(v2)),
f=(•f(v2)),
tf=(•tf(v2)),
flag=true}
t5
[•flag(v)=true]
p6
TREG
t6
[•flag(v)=false]
p7
TREG
t7
{n=(•n(v)),
f=(•f(v)),
tf=(•tf(v)),
flag=false}
t8
{n=(•n(v)),f=(•f(v)),
tf=(•tf(v)),flag=false}
p5
TREG
t9
{n=(•n(v)),
f=(•f(v)),
tf=(•tf(v)),
flag=false}
p8
TREG
[•flag(v)=false]
t10
p9
TREG
t11
@+1
p10
TREG
t12
@+1
dec(v)
p11
TREG
t14
[•n(v) = 0]
Fusion 6
ret4
TREG
t13
[•n(v) <> 0]
Fusion 1
ret1
TREG
t15
[•flag(v)=false]
Part3
Out
TREG
Part2
In
TREG


VerifyMin
Part3
In
TREG
VerifyMin
min(v)
p9
TREG
t12
[#flag(v)=true]
defsons(v)
p10
TREG
t11
[#flag(v)=false]
t13
@+1
{n=(#n(v)),
f=(#f(v)),
tf=(#tf(v)),
flag=true}
verifLoop(v)
p22
TREG
t14
[#flag(v)=false]
p11
TREG
t15
[#flag(v)=true]
decsons(v)
t27
@+1
p23
TREG
Part4
Out
TREG
t28
p24
TREG
t29
@+1
dec(v)
p26
TREG
Fusion 6
ret4
TREG
t33
[#n(v) = 0]
[#n(v) <> 0]
t32
Fusion 4
ret1
TREG
p12
TREG
t16
[#flag(v)=false]
p13
TREG
t17
@+1
Fusion 1
R
TREG
v1
pop
[(#n(v1))=
(#n(v2))]
{n=(#n(v2)),
f=(#f(v2)),
tf=(#tf(v2)),
flag=true}
v2
p
TREG
t18
[#flag(v)=true]
p12a
TREG
t18a
{n=(#n(v)),
f=(#f(v)),
tf=(#tf(v)),
flag=true}
p15
TREG
[#flag(v)=true]
t20
p16
TREG
t21
t17
TREG
p17
t22
[#flag(v)=false]
t19
{n=(#n(v)),
f=(#f(v)),
tf=(#tf(v)),
flag=false}
{n=(#n(v)),
f=(#f(v)),
tf=(#tf(v)),
flag=false}
p18
TREG
[#flag(v)=false]
t23