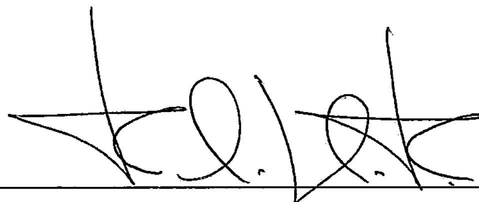
COMPILANDO RESOLUÇÃO DE PROBLEMAS PARA MINIMIZAÇÃO DE ENERGIA

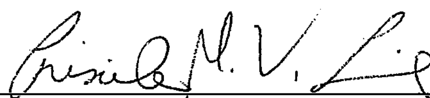
Miriam Mariela Mercedes Morveli Espinoza

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

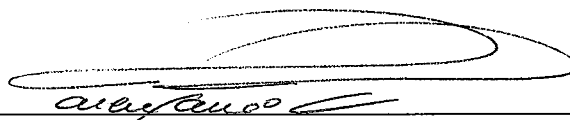
Aprovada por:



Prof. Felipe Maia Galvão França, Ph.D.



Profa. Priscila Machado Vieira Lima, Ph.D.



Profa. Marley Maria Bernardes Rebuzzi Vellasco, Ph.D.

Profa. Inês de Castro Dutra, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO DE 2006

MORVELI ESPINOZA, MIRIAM MARIELA MERCEDES

Compilando Resolucão de Problemas para Minimizacão de Energia [Rio de Janeiro] 2006

XIII, 169 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2006)

Dissertação – Universidade Federal do Rio de Janeiro, COPPE

1 - Redes de Hopfield de Alta Ordem. 2 - Simulated Annealing. 3 - Satisfatibilidade. 4 - Minimização de Energia. 5 - Plataforma SATyrus.

I. COPPE/UFRJ II. Título (série)

Aos meus pais Victoria e Melquiades
por seu grande amor e dedicação.

Agradecimentos

Agradeço em primeiro lugar a Deus, por todas as vivências que me permitiram aprender, conhecer e crescer mais a cada dia. Ao meu grande amigo Antônio, porque sem sua ajuda não houvesse sido possível chegar e permanecer aqui.

Aos meus pais Victoria e Melquiades por seu apóio moral, sentimental e econômico. E especialmente pelos seus conselhos e exemplo. Eu amo muito vocês!!.

Ao Felipe e à Priscila pela orientação durante o desenvolvimento deste trabalho e pela compreensão e apoio. À Profa Inês e à Profa. Marley, membros da banca, pelos seus conselhos e aportes finais.

Ao Karl, uma pessoa muito especial que estive nos momentos mais difíceis dando-me seu apoio e força. E claro, também esteve nos momentos mais divertidos, compartilhando alegria e bom humor.

Ao Carlos Guillen, Julio Gonzales e Javier Diaz por sua confiança e amizade. Ao Alvaro por sua ajuda quando cheguei. A Esther, Felipe, Juan Carlos, Gladys, Guina, Liliana, Luis, Saulo e Yalmar pelos momentos bons e divertidos durante a vida em república. E pelos últimos, inesquecíveis e muito divertidos e desestressantes meses, um agradecimento especial a Kely, Manuel e Edgard.

Ao meu "grande"amigo Marcelão, por escutar sempre minhas histórias e se preocupar por mim. Ao Saulo, por ter tido a coragem de ler meus primeiros intentos de escrita em português. Ao Mario Konrad por sempre tirar todas minhas dúvidas de programação, ainda sem me conhecer. Aos meus amigos brasileiros Flavia, Guilherme, Mônica e muito em especial a minha amiga Glaucia.

Aos meus amigos na distância Abigail, Gustavo, Hilda, Jessica, Liz, Mario, Miguel, Yeni e Rolf. Á minha amiga Eloisa, por sua companhia e amizade incondicional.

Finalmente, a todos aqueles que de alguma forma fizeram este período de mestrado um dos melhores da minha vida.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc)

COMPILANDO RESOLUÇÃO DE PROBLEMAS PARA MINIMIZAÇÃO DE ENERGIA

Miriam Mariela Mercedes Morveli Espinoza

Setembro/2006

Orientadores: Felipe Maia Galvão França
Priscila Machado Vieira Lima

Programa: Engenharia de Sistemas e Computação

A primeira parte deste trabalho apresenta o compilador de energia do SATyrus, uma plataforma neural que combina Redes de Hopfield de Alta Ordem, *Simulated Annealing* e satisfatibilidade para resolver problemas de otimização e qualquer outro que possa ser especificado utilizando restrições pseudo-booleanas. O compilador fornece uma linguagem para a especificação dos problemas utilizando cláusulas na Forma Normal Conjuntiva (FNC), as quais são transformadas em uma função de energia correspondente. O objetivo principal para o desenvolvimento deste compilador foi facilitar o processo de conversão e garantir que não existam erros na Função de Energia final. Na segunda parte, a ARQ-PROP-II, uma arquitetura neural para a construção de provas proposicionais utilizando o Princípio da Resolução, foi compilada e simulada utilizando o SATyrus.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

COMPILING PROBLEMS RESOLUTION FOR ENERGY MINIMIZATION

Miriam Mariela Mercedes Morveli Espinoza

September/2006

Advisors: Felipe Maia Galvão França
Priscila Machado Vieira Lima

Department: Computing and Systems Engineering

The first part of this work presents the energy compiler of SATyrus, a neural plataform that combines high-order Hopfield neural networks, simulated annealing and satisfatibility to solve optimization problems and other problems that can be specified using pseudo-boolean constraints. This compiler supplies a language for problems specification through clauses in the Conjunctive Normal Form (CNF), which are transformed into an equivalent Energy Function. The main objetive for the development of this compiler was to facilitate the conversion process and to guarantee that errors in the final Energy Function do not exist. In the second part, ARQ-PROP-II, a neural architecture for the construction of propositional proofs based on the Resolution Principle, was compiled and simulated using SATyrus.

Sumário

1 Introdução 1

1.1 Trabalhos Correlatos 2

1.2 Objetivos 3

1.3 Contribuições 3

1.4 Estrutura do Documento 4

2 Conceitos Básicos 5

2.1 Redes de Hopfield de Alta Ordem 5

2.2 Simulated Annealing 9

2.3 Satisfatibilidade 11

2.4 Minimização de Energia 12

2.4.1 Mapeamento de Satisfatibilidade para Minimização de Energia 13

2.4.2 Exemplos 15

2.4.2.1 Mapeando TSP 15

2.4.2.2 Mapeando o Problema de Coloração de Grafos . . 18

2.5 Comentários 20

3 A Plataforma SATyrus 21

3.1 Visão Geral 21

3.1.1 Arquitetura do SATyrus 21

3.1.1.1 Problema Alvo 23

3.1.1.2 Compilador 23

3.1.1.3 Simulador 23

3.1.1.4 Solução do Problema 23

3.2 Descrição do Simulador 24

3.2.1 Criação da Rede de Hopfield de Alta Ordem 24

3.2.2 O Algoritmo Simulated Annealing 25
3.2.3 Entradas do Simulador 28
3.2.4 Saídas do Simulador 28
3.3 Exemplos de Simulação 29
3.3.1 Simulando o TSP 29
3.3.2 Simulando o Problema de Coloração de Grafos 30
3.4 Comentários 33

4 O Compilador de Energia **35**
4.1 Linguagem do SATyrus 35
4.1.1 Definição das Estruturas 36
4.1.1.1 Atribuições a um Identificador 36
4.1.1.2 Matrizes de Neurônios 37
4.1.1.3 Matrizes de Valores Numéricos 39
4.1.2 Definição das Restrições 39
4.1.3 Definição das Penalidades 40
4.1.4 Exemplos 41
4.1.4.1 Código de TSP 41
4.1.4.2 Código do Problema de Coloração de Grafos 41
4.2 Visão Geral do Compilador 42
4.3 Análise Léxica 42
4.4 Análise Sintática 44
4.5 Geração da Representação Intermediária 46
4.6 Geração da Função de Energia 48
4.6.1 Criação dos Neurônios 48
4.6.2 Geração das Parcelas 50
4.7 Exemplos de Compilação 51
4.7.1 Compilando o TSP 51
4.7.2 Compilando o Problema de Coloração de Grafos 54
4.8 Comentários 54

5 Resultados: Compilando e Simulando a ARQ-PROP-II **57**
5.1 ARQ-PROP-II 57

5.2 Estruturas Neuronais da ARQ-PROP-II 58
5.3 Conjunto de Restrições da ARQ-PROP-II 61
5.4 Resultados 66
5.4.1 Teste 1 67
5.4.2 Teste 2 70
5.4.3 Teste 3 74
5.5 Comentários 74

6 Conclusões **79**

6.1 Resumo 79
6.2 Discussão 80
6.3 Trabalhos Futuros 81

Referências Bibliográficas **82**

A Manual do Compilador **86**

A.1 Estrutura do Arquivo Principal 86
A.1.1 Definição das Estruturas 86
A.1.2 Definição das Restrições 88
A.1.3 Definição das Penalidades 89
A.2 Estrutura dos Arquivos Secundários 90
A.2.1 Arquivos das estruturas que representam neurônios 90
A.2.2 Arquivos das Estruturas que não Representam Neurônios . . 91

B Código do Analizador Léxico em Flex++ **92**

C Gramática da Linguagem do SATyrus **97**

D Código Fonte da ARQ-PROP-II **102**

E Código Fonte dos Testes do Compilador **105**

F Resultados dos Testes do Compilador **151**

Lista de Figuras

2.1.1 Rede de Hopfield Binária. 6

2.1.2 Rede de Hopfield de Alta Ordem. 8

2.1.3 Exemplo de Função de Energia. 9

2.4.4 Rede de Hopfield Gerada a partir da Cláusula ϕ. 15

3.1.1 Arquitetura do SATyrus 22

3.2.2 Arquitetura do Simulador 24

3.2.3 Red de Hopfield Equivalente a F. 25

3.2.4 Estrutura da Rede de Hopfield no Simulador. 26

3.2.5 Pseudo-código de *Simulated Annealing*. 27

3.3.6 (a) Mapa Original das Cidades. (b) Mapa usado como Entrada para o SATyrus. 30

3.3.7 Saída do Simulador para o TSP. 31

3.3.8 (a) Primeiro Caminho Encontrado pelo Simulador. (b) Segundo Caminho Encontrado pelo Simulador. 31

3.3.9 Trajetoria na Função de Energia do TSP. 32

3.3.10 (a) Grafo Inicial a ser Colorido. (b) Resultado da Coloração do Grafo depois de rodar o Simulador. 32

3.3.11 Saída do Simulador para o Problema de Coloração de Grafos. . . . 33

3.3.12 Trajetoria na Função de Energia do Problema de Coloração de Grafos. 34

4.2.1 Arquitetura do Compilador. 43

4.5.2 Diagrama de Classes da Representação Intermediária. 47

4.6.3 Informação Associada a cada Parcela da Função de Energia. 50

4.7.4 Parte das Parcelas da Função de Energia do TSP. 52

4.7.5 Neurônios Gerados para uma Instância de TSP com n=5. 53

4.7.6 Partes das Parcelas da Função de Energia do Problema de Coloração de Grafos. 54
4.7.7 Neurônios Gerados para uma Instância do Problema de Coloração de Grafos com n=4. 55

5.1.1 Passos de Processo de Resolução. 58
5.2.2 Conjuntos de Neurônios da ARQ-PROP-II. 59
5.4.3 Caminho que leva à Cláusula Vazia (Teste 1). 68
5.4.4 Trajetória da Função de Energia (Teste 1). 68
5.4.5 Configuração Final da ARQ-PROP-II (Teste 1). 69
5.4.6 Caminhos que levam à Cláusula Vazia (Teste2). 71
5.4.7 Trajetória da Função de Energia (Teste 2). 71
5.4.8 Primeira Configuração Final do ARQ-PROP-II (Teste 2). 72
5.4.9 Segunda Configuração Final do ARQ-PROP-II (Teste 2). 73
5.4.10 Caminhos que levam à Cláusula Vazia (Teste3). 75
5.4.11 Trajetória da Função de Energia (Teste 3). 75
5.4.12 Primeira Configuração Final do ARQ-PROP-II (Teste 3). 76
5.4.13 Segunda Configuração Final do ARQ-PROP-II (Teste 3). 77

1.1.1 Definição das Estruturas Usadas em TSP 88
1.1.2 Definição das Restrições do TSP 90
1.1.3 Definição das penalidades do TSP 90

6.0.1 Espaço de Busca da Função de Energia (TESTE1 ate TESTE24). . 152
6.0.2 Espaço de Busca da Função de Energia (TESTES 25, 30, 31 e 36). 153
6.0.3 Espaço de Busca da Função de Energia (TESTES 26, 27, 28 e 32-34).154
6.0.4 Espaço de Busca da Função de Energia (TESTES 29 e 35). 155
6.0.5 Espaço de Busca da Função de Energia (TESTES 37, 38, 42 e 55). 157
6.0.6 Espaço de Busca da Função de Energia (TESTES 39, 40, 41, 43-46 e 52-54). 158
6.0.7 Espaço de Busca da Função de Energia (TESTES 47, 48-50, 56-59). 159
6.0.8 Espaço de Busca da Função de Energia (TESTE 51). 160
6.0.9 Espaço de Busca da Função de Energia (TESTES 61 e 62). 160
6.0.10 Espaço de Busca da Função de Energia (TESTE 63). 162

6.0.11 Espaço de Busca da Função de Energia (TESTES 64-66). 162
6.0.12 Espaço de Busca da Função de Energia (TESTE 67). 163
6.0.13 Espaço de Busca da Função de Energia (TESTE 68). 166
6.0.14 Espaço de Busca da Função de Energia (TESTE 69). 168
6.0.15 Espaço de Busca da Função de Energia (TESTE 70 e 71). 169

Lista de Tabelas

2.1 Tabela Verdade da Cláusula ϕ. 12

3.1 Tabela de Equivalência de Variável Proposicional a Neurônio. 25

4.1 Tabela de *tokens* e suas Respectivas Descrições. 45

4.2 Tabela com todos os Identificadores e Nomes dos Neurônios da Matriz $pos(3,3)$. 49

4.3 Tabela das Parcelas Geradas a partir do Padrão $\sum_{i=1}^{3}\sum_{j=1}^{3}(res[i][j]iin[i][j])$. 51

5.1 Níveis de Penalidade. 62

F.1 Tabela de Combinações para 4 Variáveis. 152

F.2 Tabela de Combinações para 3 Variáveis. 153

F.3 Tabela de Combinações para 5 Variáveis. 156

F.4 Tabela de Combinações para 2 Variáveis. 160

F.5 Tabela das Combinações para o TESTE 63. 161

F.6 Tabela de Combinações para 6 Variáveis (parte 1). 164

F.7 Tabela de Combinações para 6 Variáveis (parte 2). 165

F.8 Tabela das Combinações para o TESTE 69. 167

F.9 Tabela das Combinações para o TESTE 70. 168

Capítulo 1

Introdução

Os problemas de otimização são amplamente investigados em Ciência da Computação. Pesquisar nesta área inclui o desenvolvimento de técnicas eficientes para encontrar mínimo ou mínimos globais de uma função de muitas variáveis, chamada Função Objetivo, que avalia quão boa ou má é uma solução em um dado ponto do espaço de busca.

SATyrus [21] é uma plataforma neural para resolver problemas de otimização e que combina conceitos como Redes de Hopfield de Alta Ordem, *Simulated Annealing* [17] e satisfatibilidade [28]. Estes são especificados utilizando restrições pseudo-booleanas que descrevem seu comportamento e sua Função Objetivo. Devido à flexibilidade do sistema SATyrus, este se torna um resolvedor de qualquer problema expressável através de restrições.

O compilador de energia, desenvolvido neste trabalho, é um dos dois módulos principais do SATyrus. É fornecida uma linguagem prática para a especificação dos problemas que produz, depois do processo de compilação, uma Função de Energia que representa o espaço de busca. A precisão, exatidão e rapidez na geração desta são alguns dos principais motivos para a criação do compilador. O mínimo ou mínimos globais da Função de Energia representam a solução ou as soluções ótimas do problema. O encarregado de atingir este mínimo ou mínimos é o simulador do SATyrus desenvolvido em [24].

Para testar tanto o desempenho do SATyrus quanto o bom funcionamento do compilador de energia usamos como exemplos básicos o problema do caixeiro viajante ou TSP (*Traveling Salesperson Problem*) e o problema de coloração de grafos.

A ARQ-PROP-II, uma máquina neural de inferências proposta por Lima [19], foi também usada para avaliar o desempenho do compilador em problemas maiores e mais complexos.

Nas seções seguintes são apresentados alguns trabalhos correlatos, os objetivos principais, as contribuições e a estrutura deste trabalho.

1.1 Trabalhos Correlatos

A geração automática de uma rede neural qualquer começa com uma especificação formal desta, a qual deve ser o suficientemente expressiva. A linguagem de especificação Z [30], baseada na teoria de jogos e na lógica de primeira ordem, é utilizada para modelar e descrever sistemas de informação utilizando notação matemática. Devido à sua versatilidade, Z também tem sido usada para especificar redes neurais como pode ser visto no trabalho de Senyard [29]. Não obstante, a transformação automática de um sistema especificado em Z em um protótipo altamente confiável é um problema mais complicado. Winikoff *et. al.* [33] abordaram este problema e utilizaram programação lógica para traduzir notação Z em lógica. O problema com este esquema de transformação é que somente pode ser utilizado para um sub-conjunto de especificações.

Evans e Sulaiman introduziram em 1994 o NEUCOMP [6] um compilador seqüencial de redes neurais que possui uma linguagem procedural de alto nível para escrever programas de simulação para alguns modelos de redes neurais. NEUCOMP compila um programa escrito em forma de uma lista de especificações matemáticas de um modelo de rede neural e o transforma em um programa objeto (modelo de rede) escolhido. Alguns modelos de redes que podem ser gerados são: Multilayer/Backpropagation, Kohonen, ART (*Adaptive Resonance Theory*), Hopfield e Potts-Glass [16]. Utilizando NEUCOMP, Evans e Sulaiman [8] simularam o problema do caixeiro viajante ou TSP utilizando redes de Hopfield contínuas e redes Potts-Glass mono-camada.

Em 1996, NEUCOMP foi estendido para NEUCOMP2 [7], um compilador de redes neurais paralelas para máquinas paralelas de memória compartilhada. NEUCOMP2 possui uma rotina para detectar ciclos na versão seqüencial gerada por

NEUCOMP e depois de uma análise de dependência de dados a transforma em uma versão paralela.

A resolução de muitos problemas de otimização começa também com a especificação dos mesmos. AMPL (*A Mathematical Programming Language*) [9], desenvolvida nos Laboratórios Bell, é uma poderosa linguagem de modelagem algébrica para descrever e resolver problemas de alta complexidade, como problemas de otimização linear e não-linear. AMPL não resolve estes problemas diretamente, chama resolvedores externos apropriados para cada um deles.

Uma outra ferramenta MathSAT [2] é o resultado da integração de procedimentos de decisão matemáticos e baseados em SAT. Foi criada para problemas SMT (*Satisfiability Modulo Theories*), que podem ser vistos como uma forma estendida de satisfatibilidade proposicional, onde as proposições são ou booleanas ou restrições atômicas sem quantificadores numa teoria especifica. Depois de realizar um pré-processamento das fórmulas de entrada, MathSAT enumera as possíveis atribuições de valores-verdade a estas para finalmente verificar a consistência das teorias. Na sua última versão, MathSAT3 [3], suporta mais teorias e possui uma linguagem mais rica.

1.2 Objetivos

Os principais objetivos deste trabalho são:

- Facilitar o processo de conversão das restrições para uma função de energia equivalente.
- Garantir que não existam erros de cálculo nem procedimentais durante esta conversão.
- Obter uma função de energia livre de erros e completamente reduzida.

1.3 Contribuições

As principais contribuições deste trabalho são:

- A implementação de um compilador que permita transformar o conjunto de restrições a uma função de energia equivalente de forma rápida e garantindo a geração da função de energia de forma correta;
- A linguagem de especificação criada para escrever os problemas a serem resolvidos;
- A primeira implementação e simulação da ARQ-PROP-II, uma arquitetura neural de complexidade considerável;
- Ajustes no mapeamento do SAT para minimização de energia, utilizados na implementação do ARQ-PROP-II.

1.4 Estrutura do Documento

Este documento está estruturado em seis capítulos. O Capítulo 2 apresenta os conceitos básicos nos quais esta tese está baseada. O Capítulo 3 introduz a plataforma SATyrus e um breve resumo sobre a arquitetura e o funcionamento do simulador, um dos seus módulos principais. A descrição detalhada do compilador do SATyrus, que foi desenvolvido neste trabalho, é apresentada no Capítulo 4. A execução do sistema e os resultado obtidos dos testes são mostrados no Capítulo 5. Por último, o Capítulo 6 apresenta as conclusões, uma discussão em base a alguns trabalhos correlatos e os trabalhos futuros que podem ser desenvolvidos como continuação deste trabalho.

Capítulo 2

Conceitos Básicos

Neste capítulo revisaremos os principais conceitos nos quais esta tese se baseia. Uma descrição da estrutura, das características e da função de energia da rede de Hopfield binária e de alta ordem serão vistos na primeira seção. Na segunda seção é feita uma introdução sobre conceitos relacionados com o *Simulated Annealing* e seu funcionamento. Na Seção 2.3 veremos o conceito de satisfatibilidade e por último na Seção 2.4 descreveremos os passos necessários para mapear um problema de satisfatibilidade em um problema de minimização de energia.

2.1 Redes de Hopfield de Alta Ordem

Uma rede de Hopfield binária é composta por neurônios McCulloch-Pitts, cujos valores de entrada e saída são 0 ou 1. A rede não tem neurônios especiais de entrada ou de saída, todos são de entrada e de saída. Cada neurônio i pode estar conectado com os demais, mas não com ele mesmo, isto é ($w_{ii} = 0$). Além disso, os pesos das conexões entre pares de neurônios são simétricos, o que significa que o peso da conexão do neurônio i com o neurônio j é igual ao peso da conexão do neurônio j com o neurônio i ($w_{ij} = w_{ji}$). A Figura 2.1.1 mostra a arquitetura de uma rede de Hopfield binária de n neurônios.

A informação que entra na rede é previamente codificada e representada por um vetor de valores binários cujo número de componentes é igual ao número de neurônios que a rede tem. A entrada é aplicada simultaneamente a todos os neurônios e durante o processamento realizado pela rede estes atualizam seus estados dependendo das entradas que recebam dos outros neurônios e da função de

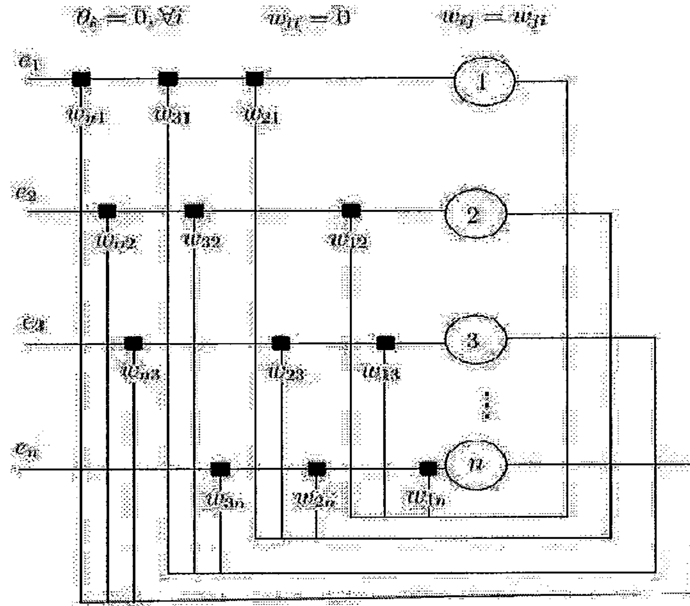


Figura 2.1.1: Rede de Hopfield Binária.

ativação. O processo de atualização dos neurônios pode ser **síncrono**, onde todos os neurônios se atualizam simultaneamente baseados nos valores produzidos no passo anterior, ou **assíncrono**, onde somente neurônios não-vizinhos atualizam seus valores simultanemente. Este processo continua até que as saídas dos neurônios alcancem um estado estável. A nova configuração dos neurônios constitui a saída da rede.

A função de ativação $f(x)$ de cada neurônio i da rede é do tipo degrau:

$$f(x) = \begin{cases} 1 & se \quad x \geq 0 \\ 0 & se \quad x < 0 \end{cases} \tag{2.1}$$

onde:

$x = \sum_{j=1}^{n} w_{ij} s_j$.

w_{ij} é o peso da conexão do neurônio i com o neurônio j.

s_j é o valor do estado do neurônio j.

n é o número de neurônios da rede.

A contribuição mais importante de Hopfield em [15] é a introdução do conceito de **Função de Energia** da rede, a qual representa todos os possíveis estados desta. Para uma rede completamente conectada a função de energia é:

$$E = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}s_is_j + \sum_{i=1}^{n}\theta_i s_i \quad (2.2)$$

onde:

w_{ij} é o peso da conexão entre os neurônios i e j.

s_i é o valor da saída do neurônio i.

s_j é o valor da saída do neurônio j.

θ_i é o limiar da função de ativação do neurônio i.

n é o número de neurônios da rede.

A propriedade principal desta função é que esta sempre decresce (ou permanece constante) durante o processamento da rede, o que garante sua convergência para um mínimo local. Entretanto a convergência para o mínimo global não é garantida.

Para garantir que a rede atinja o ótimo global, incorpora-se um comportamento estocástico aos neurônios desta. Para o processamento da rede usa-se um algoritmo chamado *Simulated Annealing* que será descrito com maior detalhe na Seção 2.2. A combinação da rede de Hopfield mais o algoritmo *Simulated Annealing* é chamada de **Redes de Hopfield Estocásticas**. A técnica de *Simulated Annealing* pode ainda ser aplicada ao aprendizado dos pesos de uma rede neural. Esta combinação chama-se **Máquina de Boltzmann** e foi proposta em [13].

Dependendo do problema que se deseja resolver, o número de neurônios envolvidos numa conexão pode ser superior a dois. Estes tipos de conexões são conhecidos como **conexões de alta ordem** ou **conexões multiplicativas**. Segundo o número de neurônios que pertencem à conexão define-se a sua **aridade**. Por exemplo, numa **conexão ternária** a aridade é 3. As características definidas para as conexões binárias são também aplicáveis às conexões de alta ordem, isto é, as conexões continuam sendo simétricas e não existe auto-influência. A rede de Hopfield que contém este tipo de conexões chama-se de **rede de Hopfield de Alta Ordem**. A Figura 2.1.2 mostra a arquitetura de uma rede de Hopfield de Alta Ordem que contém uma ligação ternaria (w_{123}).

Da mesma forma que as redes de Hopfield binárias, as redes de Hopfield de alta ordem só atingem um mínimo local. Novamente recorre-se ao algoritmo de *simulated annealing* para assegurar a convergência para o mínimo global. Em [19]

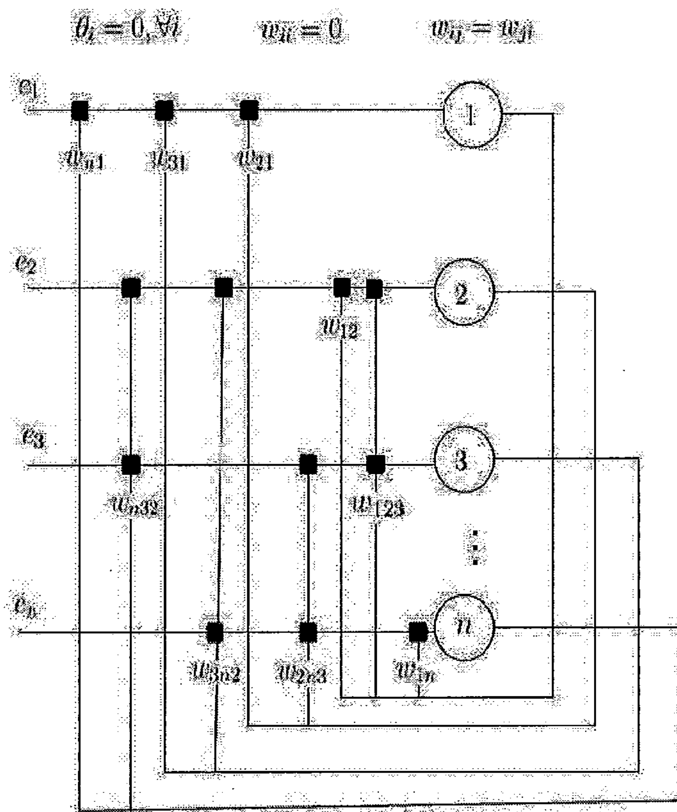


Figura 2.1.2: Rede de Hopfield de Alta Ordem.

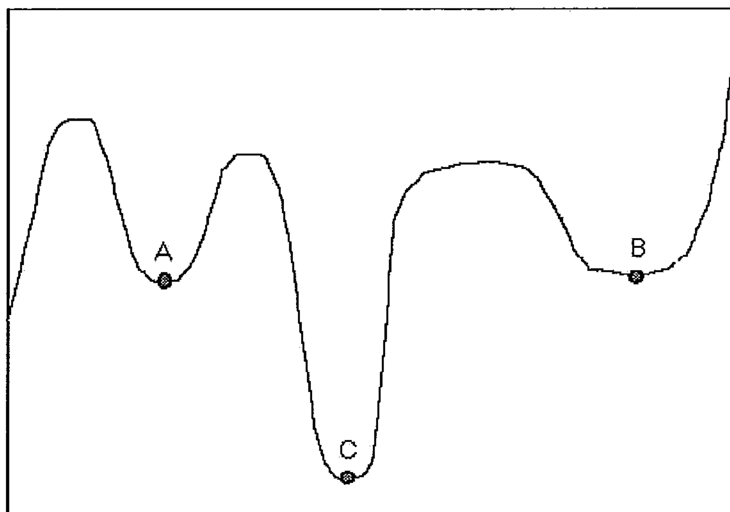


Figura 2.1.3: Exemplo de Função de Energia.

discute-se que o trabalho de Geman e Geman[12] também demonstra, com base na equivalência entre *Gibbs Random Fields* e *Markov Random Fields*, que estas atingem o ótimo global de energia. Na Figura 2.1.3 os pontos A e B são mínimos locais (redes de Hopfield binárias ou de alta ordem) e o ponto C é o ótimo global da função (redes de Hopfield estocáticas binárias ou de alta ordem).

2.2 Simulated Annealing

Em muitos problemas, o importante é encontrar não só um mínimo local, senão, o ótimo global. O *Simulated Annealing* (SA) [17] é um algoritmo probabilístico para este tipo de problema de otimização, onde encontrar o mínimo global às vezes torna-se complexo.

O nome do algoritmo vem de *annealing* em metalurgia, uma técnica que envolve esquentar e resfriar controladamente um material para aumentar o tamanho do seus cristais e para reduzir seus defeitos. O calor faz com que os átomos não fiquem estacionados na mesma posição inicial e se movimentem aleatoriamente através dos estados de uma energia mais elevada e o esfriamento devagar lhes dá mais possibilidades de encontrar configurações com energia interna mais baixa do que a inicial.

Este algoritmo pode trabalhar combinado com as redes de Hopfield binárias ou

de alta ordem para encontrar o ótimo global da função de energia destas. Fazendo uma comparação, podemos dizer que cada ponto s do espaço de busca é um estado no sistema físico, e a função $E(s)$ a ser minimizada é interpretada como a energia interna do sistema nesse estado. Conseqüentemente o objetivo é levar o sistema de um estado inicial arbitrário a um estado com a mínima energia possível. Em cada passo, o *Simulated Annealing* considera algum vizinho s' do estado atual s, e decide probabilisticamente entre mover o sistema para o estado s' ou permanecer no estado atual s. As probabilidades são escolhidas de modo que o sistema tenda a se mover para estados de energia mais baixa. Esta etapa é repetida até que o sistema alcance um estado que seja ótimo (mínimo global).

A probabilidade de fazer a transição do estado atual s para o novo estado s' é uma função $P(e, e', T)$ onde:

$e = E(s)$ é a energia no estado s,

$e' = E(s')$ é a energia no estado s',

T é um parâmetro que varia com o tempo chamado temperatura.

É importante exigir que a probabilidade P de transição não seja zero quando $e' > e$. Isto significa que o sistema pode se mover para o novo estado mesmo que a energia deste seja maior que a atual. Esta é a característica que impede que o método fique parado num mínimo local.

Por outro lado, quando a temperatura vai caindo (ou seja T se aproxima de zero) a probabilidade $P(e, e', T)$ deve tender a zero se $e' > e$ e a um valor positivo se $e' < e$. Desta forma, para valores suficientemente pequenos de T, o sistema favorecerá cada vez mais os movimentos que vão para baixo (para valores baixos de energia) e evitará aqueles que vão para acima. Quando T chega a zero, o algoritmo de SA vira o algoritmo de busca gulosa, o qual se move só se a energia do novo estado é mais baixa.

A função P é geralmente escolhida de modo que a probabilidade de aceitar um movimento diminua quando a diferença $e' - e$ aumente:

$$\begin{cases} P_{net_i}(s_i = 1) = \frac{1}{1+e^{(-net_i)/T}} \\ \\ P_{net_i}(s_i = 0) = \frac{e^{(-net_i)/T}}{1+e^{(-net_i)/T}} \end{cases} \tag{2.3}$$

onde:

$net_i = (\sum w_{ij} s_j(t)) - \theta_i$

θ_i é o limiar do neurônio i

T é a temperatura ($T >= 0$)

Analisando a função de probabilidade P, fica claro que o estado s depende crucialmente de T (temperatura). A temperatura T começa com um valor muito alto e vai decrescendo de acordo com algum *annealing schedule*, o qual é especificado pelo usuário, até chegar a $T = 0$ ou quase zero. A escolha do *annealing schedule* é muito importante para garantir o êxito do algoritmo e atingir o mínimo global de energia.

2.3 Satisfatibilidade

O problema da satisfatibilidade booleana (SAT) [28] é um problema de decisão que consiste em determinar se existe uma atribuição de valores VERDADEIROS ou FALSOS aos literais (átomos positivos ou negativos) L_i de uma cláusula C que a tornarão verdadeira, isto é, que a satisfarão. Esta é composta pela conjunção de fórmulas proposicionais, as quais por sua vez contêm disjunções de literais, ou seja, a cláusula está na *Forma Normal Conjuntiva* (FNC). Por exemplo:

Seja: $\phi = (q \vee s) \wedge (p \vee \neg q) \wedge (s \vee \neg p)$ uma cláusula na FNC.

Na Tabela 2.1 observa-se que as linhas 5, 6 e 8 satisfazem a cláusula ϕ, de onde podemos concluir que esta é satisfatível nas atribuições FFV, VFV e VVV de p, q e s respectivamente.

Em 1971 Stephen Cook [4] demonstrou que SAT pertence à classe de problemas NP-Completos. Nesta forma o problema SAT não trata somente de decidir se uma cláusula é verdadeira ou não para uma combinação de literais dada, como também de encontrar alguma atribuição de valores VERDADEIROS ou FALSOS tal que maximize o número de fórmulas satisfeitas. Em outras palavras, trata-se de um problema de otimização e não só de decisão, já que se deve encontrar a melhor solução possível.

A importância e o interesse de resolver SAT é porque tem relação com diferentes problemas, e encontrar a solução representaria poder atacar aplicações em variados campos, até agora difíceis de tratar. Entre estes problemas estão por exemplo a

p	q	s	$q \vee s$	$p \vee \neg q$	$s \vee \neg p$	$(q \vee s) \wedge (p \vee \neg q) \wedge (s \vee \neg p)$
F	F	F	F	V	V	F
V	F	F	F	V	F	F
F	V	F	V	F	V	F
V	V	F	V	V	F	F
F	F	V	V	V	V	V
V	F	V	V	V	V	V
F	V	V	V	F	V	F
V	V	V	V	V	V	V

Tabela 2.1: Tabela Verdade da Cláusula ϕ.

planificação, os circuitos Hamiltonianos [5], a coloração de grafos [31], a criptologia, a satisfação de restrições, etc.

2.4 Minimização de Energia

Minimização de energia é uma técnica para modelar o comportamento de certa classe de redes neurais, tal como a rede de Hopfield, da qual já tratamos na Seção 2.1 e dizemos também como a função de energia desta é a encarregada de levar a rede a uma saída estável ante qualquer entrada dada à rede. Mas as entradas desta nem sempre são conjuntos de padrões a serem aprendidos, algumas vezes são problemas descritos usando **restrições** as quais devem ser escritas na Forma NormalC. Neste caso é preciso mapear o conjunto de restrições que descrevem o problema para uma rede neural equivalente onde o mínimo (ou mínimos) de energia global serão as soluções do problema.

Tipos de restrições

As restrições que definem um problema se dividem em dois grupos:

- *Restrições de integridade*: usadas para justificar a conversão de disjunções à conjunção de disjunções;
- *Restrições de otimalidade*: representam o que se deseja otimizar e não precisam conversão nenhuma.

Penalidade das restrições

Cada uma das restrições deve ter associada uma penalidade expressa através de uma constante multiplicativa. O cálculo das penalidades tem que ser feito em função das penalidades de menor prioridade e se calcula da seguinte forma:

- O valor da penalidade de nível 0, é 1. Existem problemas, o TSP por exemplo, que precisam de outros valores (distâncias neste caso) para descrever suas restrições de otimalidade; então o valor da penalidade mais baixa será o valor máximo dentre estes.

- O valor das penalidade de nível 1 se calcula *:

  α =[(número de cláusulas de menor prioridade $\times$ a maior penalidade delas) $+\epsilon$], onde ϵ é um valor bem pequeno.

- O valor das penalidades de nível 2 se calcula:

  $\beta =$ [(número de penalidades $\alpha \times \alpha$) + (número de cláusulas de menor prioridade $\times$ a maior penalidade delas) $+\epsilon$], ou seja:

  $\beta =$ [(número de penalidades $\alpha \times \alpha$) $+\alpha + \epsilon$].

- Por último, para calcular o valor das penalidades a partir do nível 3 se segue o último padrão.

Exemplos de problemas que podem ser definidos mediante restrições são o problema do caixeiro viajante, mais conhecido como **TSP** (*Travelling Salesperson Problem*), e **coloração de grafos**, os quais veremos mais adiante.

2.4.1 Mapeamento de Satisfatibilidade para Minimização de Energia

Pinkas [25] demonstrou que o problema de encontrar o mínimo global de uma função de energia é equivalente a resolver o problema de satisfatibilidade booleana em lógica proposicional. Podemos dizer então que para cada cláusula proposicional existe uma função de energia equivalente e vice-versa.

*Os nomes α, β e γ serão utilizados neste trabalho para nos referir aos níveis de penalidades

Para converter uma cláusula proposicional em Forma Normal Conjuntiva para sua função de energia equivalente temos que utilizar a função H descrita em [26] para mapeá-la no conjunto $\{0, 1\}$:

- $H(true) = 1$
- $H(false) = 0$
- $H(\neg p) = 1 - H(p)$
- $H(p \wedge q) = H(p) \times H(q)$
- $H(p \vee q) = H(p) + H(q) - H(p \wedge q)$

Seja: $\phi = \bigwedge_i \phi_i$ uma cláusula na Forma Normal Conjuntiva.
onde:

$\bigwedge_i \phi_i = \bigvee_j p_{ij}$ é a disjunção i dos literais p_{ij}.

A energia E está associada a $H(\neg\phi)$, onde:

$\neg\phi = \bigvee_j \neg p_{ij}$

e

$\neg p_{ij} = \bigwedge_j \neg p_{ij}$

Não obstante, se calcularmos a energia com a função H original, só teremos dois valores resultantes: 0 se a solução é encontrada e 1 caso contrário. Pinkas [26] também propõe usar a seguinte função H^* onde E pode ter mais valores que representam o número de cláusulas que não são satisfeitas pela atribuição de valores VERDADEIROS ou FALSOS.

Se:

$$E = H^*(\neg\phi) = \sum_i H(\neg\phi_i)$$

então,

$$E = \sum_i H(\bigwedge_j \neg p_{ij}) = \sum_i \prod_j H(\neg p_{ij}) \tag{2.4}$$

onde $H(p)$ será referido como p.

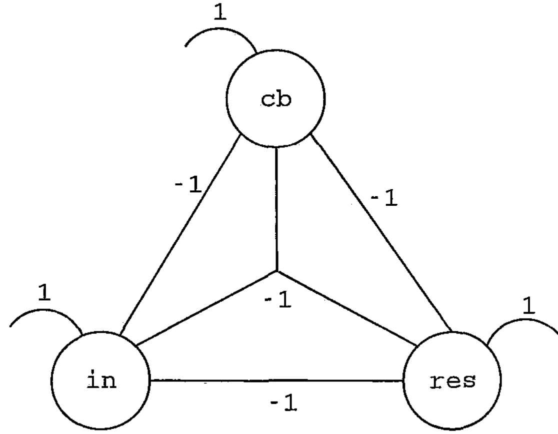


Figura 2.4.4: Rede de Hopfield Gerada a partir da Cláusula ϕ.

A função de energia resultante é a representação matemática da rede neural. Cada variável proposicional representa um neurônio da rede. O coeficiente de um produto de variáveis proposicionais representa o peso que liga os ditos neurônios. O coeficiente de um termo linear representa seu limiar e por último, os termos constantes não precisam ser representados.

Exemplo:

$$\begin{aligned}
\phi &= in \lor cb \lor res \\
E &= H(\neg(in \lor cb \lor res)) \\
&= H(\neg in \land \neg cb \land \neg res) \\
&= (1 - in) * (1 - cb) * (1 - res) \\
&= 1 - in - cb + in * cb - res + in * res + cb * res - in * cb * res
\end{aligned}$$

A rede resultante da Figura 2.4.4 possui não só conexões binárias, mas também conexões de alta ordem.

2.4.2 Exemplos

A seguir dois exemplos, extraidos de [22], que mostram o processo de mapeamento.

2.4.2.1 Mapeando TSP

Dado um número n de cidades e os custos de viajar de uma para a outra, o problema do caixeiro viajante (TSP) é encontrar o caminho de menor custo tal que se visite cada cidade exatamente uma vez e se retorne à cidade inicial.

A rede se define através de uma matriz de $n \times n$ neurônios binários pos_{ij} onde i representa a cidade e j a posição da cidade na viagem. Para forçar o retorno à cidade inicial duplicamos esta e fazemos com que a distancia entre a cidade inicial e sua cópia seja 0. Devido a isto o valor de n é igual ao número de cidades mais um.

Os quantificadores usados são para dar uma maior expressividade às restrições e depois tornam-se operadores lógicos. Isto não significa que se use lógica de primeira ordem nas restrições.

As restrições necessárias para definir o problema de TSP são:

Restrições de integridade:

(i) Todas as cidades devem fazer parte da viagem.

$\forall i, \exists j \mid 1 \leq i \leq n,\ 1 \leq j \leq n : (pos_{ij})$

Então, $\varphi_1 = \bigwedge_i (\bigvee_j (pos_{ij}))$ (2.5)

(ii) Duas cidades não podem ocupar a mesma posição na viagem.

$\forall i,\ \forall j,\ \forall k \mid 1 \leq i \leq n,\ 1 \leq j \leq n,\ 1 \leq k \leq n;\ i \neq k : \neg(pos_{ij} \wedge pos_{kj})$

Então, $\varphi_2 = \bigwedge_i \bigwedge_j \bigwedge_{k \neq i} \neg(pos_{ij} \wedge pos_{kj})$ (2.6)

(iii) Uma cidade não pode ocupar mais de uma posição na viagem.

$\forall i,\ \forall j,\ \forall k \mid 1 \leq i \leq n,\ 1 \leq j \leq n,\ 1 \leq k \leq n;\ j \neq k : \neg(pos_{ij} \wedge pos_{ik})$

Então, $\varphi_3 = \bigwedge_i \bigwedge_j \bigwedge_{k \neq j} \neg(pos_{ij} \wedge pos_{ik})$ (2.7)

Restrições de otimalidade:

Deve-se encontrar o caminho de custo mínimo, minimizando a soma de distâncias entre:

(iv) Uma cidade i e a cidade $i+1$ seguinte na viajem.

$\forall i,\ \forall j,\ \forall k \mid 1 \leq i \leq n,\ 1 \leq j \leq n-1,\ 1 \leq k \leq n;\ i \neq k : dist_{ik}(pos_{ij} \wedge pos_{k(j+1)})$

Então, $\varphi_4 = \bigwedge_i \bigwedge_{j \neq k} \bigwedge_{j<n} dist_{ik}(pos_{ij} \wedge pos_{k(j+1)})$ (2.8)

(v) Uma cidade i e a cidade $i-1$ anterior na viajem.

$\forall i,\ \forall j,\ \forall k \mid 1 \leq i \leq n,\ 2 \leq j \leq n,\ 1 \leq k \leq n;\ i \neq k : dist_{ik}(pos_{ij} \wedge pos_{k(j-1)})$

Então, $\varphi_5 = \bigwedge_i \bigwedge_{j \neq k} \bigwedge_{j>1} dist_{ik}(pos_{ij} \wedge pos_{k(j-1)})$ (2.9)

As penalidades WTA (*Winner-Takes All*) são as de valor mais alto e as representaremos com γ. Em TSP as restrições (ii) e (iii) são de tipo WTA. As restrições de otimalidade (iv) e (v) são as que tem o valor de penalidade mais baixo e para as demais restrições de integridade (i) o valor da penalidade é representado por α. O cálculo das penalidades para o TSP se resume em:

$$\begin{cases} dist = max\{dist_{ij}\} \\ \alpha = ((n^3 - 2n^2 + n) \times dist) + \epsilon \\ \gamma \quad = ((n^2 + 1) \times \alpha) + \epsilon \end{cases}$$

Função de energia equivalente

Para mapear o conjunto de cláusulas de TSP à sua função de energia equivalente usamos a função H^* anteriormente definida:

$$E_i \;=\; \alpha H^*_{WTA}(\neg\varphi_1) \;+\; \gamma H^*(\neg\varphi_2) \;+\; \gamma H^*(\neg\varphi_3) \tag{2.10}$$

Se $\quad \varphi_1 = \bigwedge_i(\bigvee_j(v_{ij}))$, então: $\quad \neg\varphi_1 = \bigvee_i(\bigwedge_j(\neg v_{ij}))$

$$\begin{aligned} H^*(\neg\varphi_1) &= \sum_{i=1}^{n} H(\textstyle\bigwedge_j(\neg v_{ij})) \\ &= \sum_{i=1}^{n} \prod_{j=1}^{n} H(\neg v_{ij}) \\ &= \sum_{i=1}^{n} \prod_{j=1}^{n}(1 - v_{ij}) \end{aligned}$$

Devido às restrições WTA $\neg\varphi_1$ vira:

$$H^*_{WTA}(\neg\varphi_1) \;=\; \sum_{i=1}^{n}\sum_{j=1}^{n}(1 - v_{ij}) \tag{2.11}$$

Se $\quad \varphi_2 = \bigwedge_i \bigwedge_j \bigwedge_{k\neq i} \neg(v_{ij} \wedge v_{kj})$, então:

$$\begin{aligned} H^*(\neg\varphi_2) &= \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1,k\neq i}^{n} H(v_{ij} \wedge v_{kj}) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1,k\neq i}^{n} v_{ij}v_{kj} \end{aligned} \tag{2.12}$$

Se $\quad \varphi_3 = \bigwedge_i \bigwedge_j \bigwedge_{k\neq j} \neg(v_{ij} \wedge v_{ik})$, então:

$$\begin{aligned} H^*(\neg\varphi_3) &= \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1,k\neq j}^{n} H(v_{ij} \wedge v_{ik}) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1,k\neq j}^{n} v_{ij}v_{ik} \end{aligned} \tag{2.13}$$

$$E_o \;=\; H^*(\neg\varphi_4) \;+\; H^*(\neg\varphi_5) \tag{2.14}$$

Se $\varphi_4 = \bigwedge_i \bigwedge_{j<n} \bigwedge_{k \neq j} dist_{ik}(v_{ij} \wedge v_{k(j+1)})$, então:

$$H^*(\varphi_4) = \sum_{i=1}^{n}\sum_{i=1}^{n}\sum_{j=1}^{n-1}\sum_{k=1,k\neq j}^{n} dist_{ik} H(v_{ij} \wedge v_{k(j+1)})$$

$$= \sum_{i=1}^{n}\sum_{i=1}^{n}\sum_{j=1}^{n-1}\sum_{k=1,k\neq j}^{n} dist_{ik} v_{ij} v_{k(j+1)}) \quad (2.15)$$

Se $\varphi_5 = \bigwedge_i \bigwedge_{j>1} \bigwedge_{k \neq j} dist_{ik}(v_{ij} \wedge v_{k(j-1)})$, então:

$$H^*(\varphi_5) = \sum_{i=1}^{n}\sum_{i=1}^{n}\sum_{j=2}^{n}\sum_{k=1,k\neq j}^{n} dist_{ik} H(v_{ij} \wedge v_{k(j-1)})$$

$$= \sum_{i=1}^{n}\sum_{i=1}^{n}\sum_{j=2}^{n}\sum_{k=1,k\neq j}^{n} dist_{ik} v_{ij} v_{k(j-1)}) \quad (2.16)$$

A função final é: $E = E_i + E_o$ (2.17)

2.4.2.2 Mapeando o Problema de Coloração de Grafos

Dado um grafo G de n vértices, o problema de coloração de grafos consiste em pintar cada um dos vértices utilizando o menor número de cores possíveis, considerando que vértices vizinhos devem ter cores diferentes e cada vértice deve ser pintado com uma só cor.

A rede é composta por 3 matrizes: (i) a matriz vc de $n \times n$ neurônios binários vc_{ik} onde i representa o vértice e k o índice da cor atribuída ao vértice i, (ii) a matriz col de $1 \times n$ neurônios binários col_k, e (iii) a matriz $neigh$ de $n \times n$ neurônios binários $neigh_{ij}$ usada para representar que o neurônio i é vizinho do neurônio j.

Da mesma forma que em TSP, tem-se dois grupos de restrições:

Restrições de integridade:

(i) Cada vértice deve ter uma cor associada.
$\forall i,\ \exists k \mid 1 \leq i \leq n,\ 1 \leq k \leq n : (vc_{ik})$
Então, $\varphi_1 = \bigwedge_i (\bigvee_j (v_{ik}))$ (2.18)

(ii) Dois vértices vizinhos não podem ter a mesma cor.
$\forall i,\ \forall j\ \forall k \mid 1 \leq i \leq n,\ 1 \leq j \leq n,\ 1 \leq k \leq n; i \neq j : \neg neigh_{ij} \vee (vc_{ik} \wedge vc_{jk})$
Então, $\varphi_2 = \bigwedge_i \bigwedge_{j \neq i} \bigwedge_k (\neg neigh_{ij} \vee (vc_{ik} \wedge vc_{jk}))$ (2.19)

(iii) Um vértice não pode ter mais de uma cor.
$\forall i,\ \forall k\ \forall l \mid 1 \leq i \leq n,\ 1 \leq j \leq n,\ 1 \leq l \leq n; k \neq l : \neg(vc_{ik} \wedge vc_{il})$

Então, $\varphi_3 = \bigwedge_i \bigwedge_k \bigwedge_{l \neq k} \neg(vc_{ik} \wedge vc_{il})$ (2.20)

(iv) Se a cor k é associada a um vértice em vc então o neurônio k na matriz *col* deve ser ativada.

$\forall i,\ \forall k \ \mid 1 \leq i \leq n,\ 1 \leq k \leq n : (\neg vc_{ik} \wedge c_k)$

Então, $\varphi_4 = \bigwedge_i \bigwedge_k (\neg vc_{ik} \wedge c_k)$ (2.21)

Restrições de otimalidade:

(v) O número de cores utilizados

$\forall k \ \mid 1 \leq k \leq n : c_k$

Então, $\varphi_5 = \bigwedge_k c_k$ (2.22)

As penalidades associadas são α às restrições (i), (ii) e (iv), γ à (iii) e a penalidade mais baixa à restrição de otimalidade. O cálculo das penalidades é:

$$\begin{cases} \alpha & = (n \times 1) + \epsilon \\ \gamma & = ((n^3 + n^2 + 1) \times \alpha) + \epsilon \end{cases}$$

Função de energia equivalente

A energia final é resultado da soma das energias das restrições de integridade E_i e da energia da restrição de otimalidade E_o:

$$E_i = \alpha[H^*_{WTA}(\neg\varphi_1) + H^*(\neg\varphi_2) + H^*(\neg\varphi_4)] + \gamma[H^*(\neg\varphi_3)] \quad (2.23)$$

$$E_o = H^*(\neg\varphi_5) \quad (2.24)$$

Como no exemplo do TSP utilizamos a função H^* e o resultado da função de energia E é:

$$H^*(\neg\varphi_1) = \sum_{i=1}^{n} \sum_{k=1}^{n} (1 - vc_{ik}) \quad (2.25)$$

$$H^*(\neg\varphi_2) = \sum_{i=1}^{n} \sum_{j=1, j\neq i}^{n} \sum_{k=1}^{n} (vc_{ik} vc_{jk} neigh_{ij}) \quad (2.26)$$

$$H^*(\neg\varphi_3) = \sum_{i=1}^{n} \sum_{k=1}^{n} \sum_{l=1, l\neq k}^{n} (vc_{ik} vc_{il}) \quad (2.27)$$

$$H^*(\neg\varphi_4) = \sum_{i=1}^{n}\sum_{k=1}^{n} vc_{ik}(1 - c_k) \qquad (2.28)$$

$$H^*(\varphi_5) = \sum_{k=1}^{n}(c_k) \qquad (2.29)$$

A função final é: $E = E_i + E_o$ (2.30)

2.5 Comentários

Este capítulo definiu as características e o funcionamento das redes de Hopfield (binárias e de alta ordem) e o *Simulated Annealing*. Introduziu o conceito de **Máquina de Boltzmann**, que resulta da união do algoritmo Simulated Annealing mais neurônios estocásticos. Esta combinação assegura a convergência para o ótimo global da função de energia da rede.

Definiu-se também o problema de satisfatibilidade (SAT) e a importância de resolvê-lo, devido a sua relação com problemas de diversos campos que podem ser representados usando SAT. Por último, mostrou-se os passos necessários para transformar um problema SAT para um problema de minimização de energia, usando a função H^*.

O próximo capítulo mostrará os módulos e as características da plataforma SATyrus, uma arquitetura neural criada para resolução de problemas SAT.

Capítulo 3

A Plataforma SATyrus

Neste capítulo definiremos e descreveremos a plataforma SATyrus. Na primeira seção se detalha a arquitetura e o funcionamento do SATyrus. Na Seção 2 se descreve os principais módulos do simulador e os algoritmos utilizados. Por último, na Seção 3 serão mostrados dois exemplos de simulação.

3.1 Visão Geral

SATyrus [21] é uma arquitetura neural criada para resolver problemas de otimização que são modelados usando restrições escritas em lógica proposicional, mais precisamente em Forma Normal Conjuntiva. SATyrus é baseada nos conceitos de redes neurais, satisfatibilidade (SAT) e minimização de energia. Depois de escrever a modelagem do problema alvo na linguagem de especificação de problemas do SATyrus, esta passa por um processo de compilação. Neste processo, o compilador converte o conjunto de restrições para uma função de energia equivalente. A função de energia é convertida numa rede estocástica de Hopfield de alta ordem que atua como um otimizador para encontrar o mínimo global.

3.1.1 Arquitetura do SATyrus

O SATyrus consta de 2 módulos básicos: um compilador e um simulador. Na Figura 3.1.1 mostramos as entradas e saídas do sistema assim como o fluxo de dados entre estas e os módulos principais. Posteriormente descreveremos os módulos básicos e as entradas e saídas de cada um deles.

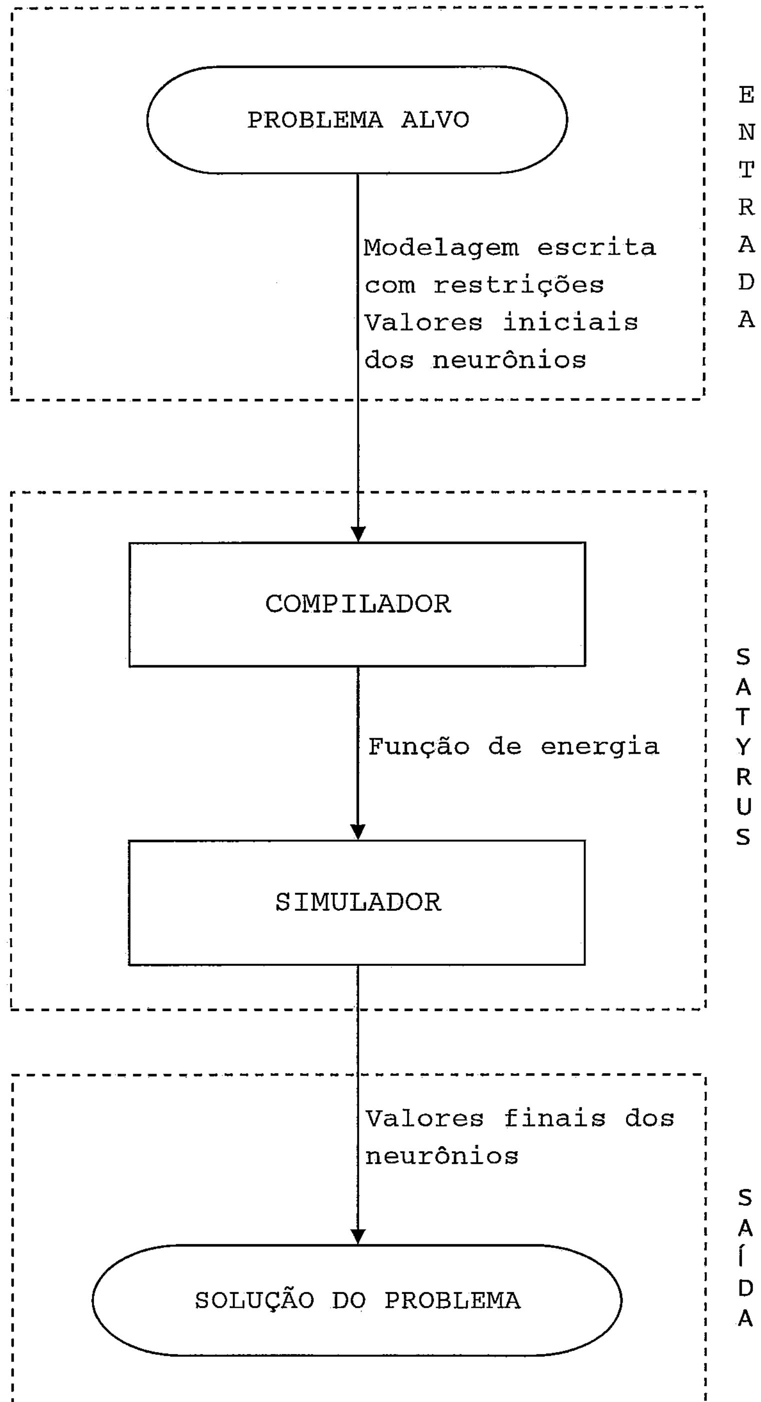


Figura 3.1.1: Arquitetura do SATyrus

3.1.1.1 Problema Alvo

O problema alvo, ou seja o problema que desejamos resolver, é qualquer um que possamos modelar usando restrições. Os problemas de otimização combinatória, como TSP e o problema de coloração de grafos, são bons exemplos para este tipo de modelagem. Como veremos adiante, a ARQ-PROP-II, uma arquitetura neural que realiza inferências proposicionais, também pode ser definida usando restrições, o que a torna um bom problema alvo a ser resolvido pelo SATyrus.

3.1.1.2 Compilador

O compilador recebe como entrada um arquivo principal, que contém basicamente o problema alvo modelado usando restrições que estão escritas na Forma Normal Conjuntiva, e arquivos secundários que servem para inicializar as matrizes de neurônios. Sobre o arquivo principal o compilador realiza as análises léxica, sintática, e extrai delas informações. Estas são usadas juntamente com os dados dos arquivos secundários para gerar uma função de energia cuja minimização corresponderá a uma solução do problema alvo. A arquitetura, funcionamento e as características principais do compilador serão vistas com maior detalhe no Capitulo 4.

3.1.1.3 Simulador

O simulador, que será descrito com mais detalhe na Seção 2 deste capítulo, recebe como entrada a função de energia gerada pelo compilador a partir da qual gera a rede neural equivalente. Usando o algoritmo de *Simulated Annealing*, percorre o espaço de soluções até encontrar o mínimo global. A saída do simulador é a solução do problema.

3.1.1.4 Solução do Problema

Ao ser atingido o mínimo global da função de energia, os valores dos neurônios da rede nesse ponto representam a solução do problema. Algumas vezes o problema tem mais de um mínimo global, neste caso o simulador é capaz de encontrar também as outras soluções.

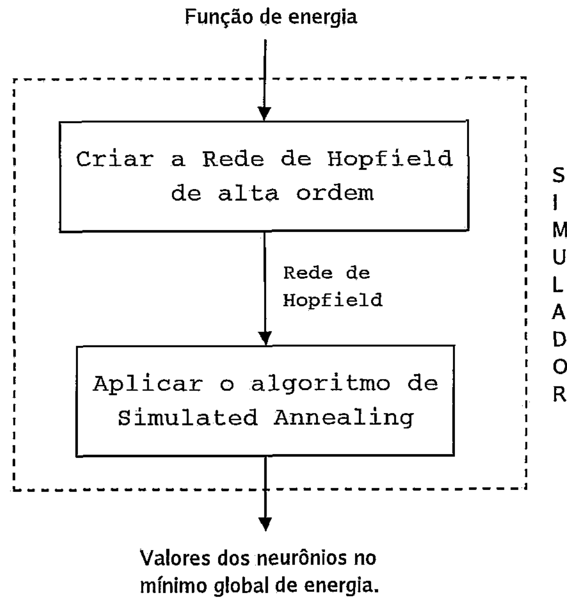


Figura 3.2.2: Arquitetura do Simulador

3.2 Descrição do Simulador

O simulador, desenvolvido em [24], é uma rede de Hopfield de alta ordem de neurônios binários que utiliza o algoritmo de *Simulated Annealing* para atingir um ou mais ótimos globais. Seu objetivo é levar a rede de um estado de entrada arbitrário para um estado final que representa a solução ótima do problema. Em outras palavras, minimiza a função de energia da rede até convergir para o mínimo global. Na Figura 3.2.2 se mostra a arquitetura do simulador, que será explicada mais detalhadamente a seguir.

3.2.1 Criação da Rede de Hopfield de Alta Ordem

Para se armazenar a rede de Hopfield de alta ordem, o simulador utiliza uma estrutura exemplificada na Figura 3.2.4. Observemos que a rede é uma lista de neurônios, onde cada neurônio i além de seus atributos próprios também possui uma lista de pesos associada. Cada peso tem associada outra lista chamada de vizinhança, que guarda os identificadores dos neurônios vizinhos ao neurônio i. Para ser vizinho deve existir uma conexão não-nula entre neurônios.

Vejamos um exemplo de como a função de energia é convertida nesta estrutura:

Variável proposicional	Neurônio
p	N1
q	N2
r	N3
s	N4

Tabela 3.1: Tabela de Equivalência de Variável Proposicional a Neurônio.

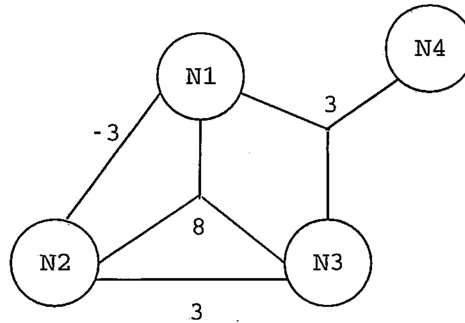


Figura 3.2.3: Red de Hopfield Equivalente a F.

Seja $F = 3pq - 3prs - 8qpr - 3qr$ a função de energia gerada pelo compilador. Como vimos no Capítulo 2, cada uma das variáveis proposicionais é mapeada em um neurônio. A Tabela 3.1 mostra o mapeamento das variáveis proposicionais p, q, r e s para os seus respectivos neurônios.

A Figura 3.2.3 mostra a rede neural resultante e na Figura 3.2.4 se mostra a estrutura que o simulador cria para F.

3.2.2 O Algoritmo Simulated Annealing

Como mencionamos no Capítulo 2, a rede de Hopfield converge para mínimos locais. Não obstante, o objetivo principal do simulador é encontrar mínimos globais. A solução é fazer com que os neurônios da rede tenham um comportamento estocástico, isto é, aplicar o algoritmo *Simulated Annealing* na rede, para que o ótimo global seja encontrado.

Antes de detalhar os passos do algoritmo *Simulated Annealing* utilizado, faremos algumas definições importantes que ajudarão na melhor compreensão do mesmo.

Seja $N = \{n_1, n_2, ..., n_n\}$ o conjunto dos neurônios da rede e $V = \{v_1, v_2 ... v_n\}$ um conjunto de variáveis aleatórias. Associamos a cada nó n_i uma variável aleatória

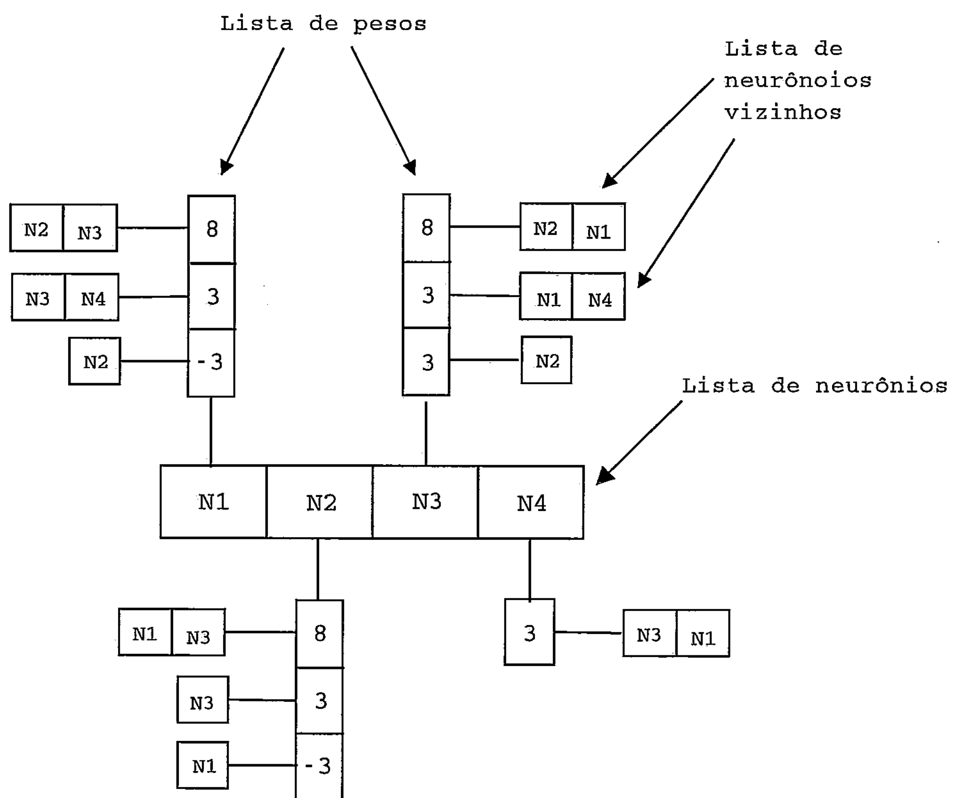


Figura 3.2.4: Estrutura da Rede de Hopfield no Simulador.

```
T = T0
while T ≥ Tf
  begin
    for i = 1 to n do
      Atualizar o estado do neurônio i.
    T = αT
  end
```

Figura 3.2.5: Pseudo-código de *Simulated Annealing.*

v_i, cujos valores são obtidos de um domínio finito comum $D^n = \{0,1\}$. Cada v_i representa o estado do neurônio n_i e cada elemento de D^n é um possível estado dele. Por último, para cada v_i definimos um conjunto de vizinhos $Q(v_i)$. v_i e seus vizinhos formam uma vizinhança homogênea, isto é, se $v_j \in Q(v_i)$ então $v_i \in Q(v_j)$.

A Figura 3.2.5 mostra o algoritmo de *Simulated Annealing* utilizado pelo simulador.

Atualização do estado do neurônio i

Para atualizar o estado do neurônio i o simulador segue os seguintes passos:

1. Calcular a probabilidade de aceitação da troca de estado. Para tal utiliza as equações 2.3:

   $$P_{net_i}(s_i = 1) = \frac{1}{1+e^{(-net_i)/T}} \quad e \quad P_{net_i}(s_i = 0) = \frac{e^{(-net_i)/T}}{1+e^{(-net_i)/T}};$$

   onde:

   $net_i = (\sum w_{ij} v_j(t)) - \theta_i$.

   θ_i é o limiar do neurônio i.

   T é o parâmetro temperatura ($T \geq 0$).

2. Verificar a probabilidade de trocar, ou não, de estado. Para isto se define uma variável randômica, cujo valor está entre 0 e 1. Se o estado atual do neurônio $n_i = 0$ e a probabilidade de trocar para 1 é maior que a variável probabilística, então o estado de n_i muda para 1, caso contrário permanece em 0. Se o estado atual do neurônio $n_i = 1$ e a probabilidade de trocar para

0 é maior que a variável probabilística, então o estado de n_i muda para 0, caso contrário permanece em 1.

3. Enviar uma messagem a todos os vizinhos do neurônio i com o novo estado deste.

4. Calcular a função de energia da rede.

Este algoritmo termina quando se atinge a temperatura final T_f. Não obstante, o simulador dá a opção de recomeçar o algoritmo caso se tenha ou não atingido o mínimo global. Os valores finais dos estados dos neurônios se tornam os novos valores iniciais destes. Isto permite atingir um novo mínimo global, se um já houvesse sido encontrado, ou encontrar o mínimo global esperado, se nas rodadas anteriores só mínimos locais houvessem sido atingidos.

O funcionamento e as características do simulador do SATyrus são apresentadas com muito mais detalhe em [24].

3.2.3 Entradas do Simulador

A função de energia, que é a saída do compilador, pode ser vista como uma entrada para o simulador gerada computacionalmente. Além desta, o simulador também recebe duas entradas provenientes do usuário para personalizar o funcionamento do algoritmo. Estas entradas são:

(i) O fator de atualização da temperatura T, que determina a velocidade de resfriamento desta. O valor padrão deste parâmetro é $0,99$;

(ii) O valor da velocidade do *loop* é um parâmetro que serve para modificar o valor do fator de atualização da temperatura quando mais de uma rodada é feita. O valor padrão deste parâmetro é $0,9$.

3.2.4 Saídas do Simulador

A saída principal do simulador são os valores dos neurônios que constituem a solução do problema, sem que isto signifique que são os últimos valores (da iteração final) atribuidos aos neurônios. Acontece que o simulador só pára quando atinge a temperatura final T_f. Existe então a possibilidade de se passar antes

pelo mínimo global e não parar nele. A solução para isto é guardar os valores dos neurônios nesse ponto para depois exibí-los junto com os valores da última iteração, mostrando também os valores de suas respectivas energias.

Outra saída secundária, mas também importante, é um arquivo com os valores da função de energia em cada iteração. Podemos gerar um gráfico a partir destes dados que mostra o comportamento da função de energia.

3.3 Exemplos de Simulação

No Capítulo 2 vimos a modelagem do TSP e do problema de coloração de grafos utilizando restrições, e como estas são convertidas para um problema de minimização de energia. O passo seguinte é gerar a função de energia equivalente, processo que é feito pelo compilador do SATyrus. A seguir mostramos os resultados obtidos do simulador depois de otimizar a função de energia entrante.

3.3.1 Simulando o TSP

Para mostrar o funcionamento do simulador em TSP, na Figura 3.3.6 (a) mostramos o mapa das cidades com suas respectivas distâncias e na Figura 3.3.6 (b) o mapa que o SATyrus recebe como entrada. Como foi explicado na modelagem do TSP, este último mapa tem uma cidade a mais, que é a réplica da cidade inicial e representa a cidade final. As cidades inicial e final estão representadas por um círculo duplo.

A Figura 3.3.7 mostra os resultados obtidos pelo simulador. A seguir descrevemos e interpretamos esta saída:

- A matriz pos_{ij} foi definida na modelagem do TSP no Capítulo 2. Sabemos então que i representa a cidade e j a posição desta na viagem.

- A primeira coluna do lado do nome dos neurônios é a melhor configuração da rede encontrada e a segunda coluna é a configuração da iteração final. Neste caso, ambas têm o mesmo valor de energia o que significa que o problema tem duas soluções, ou seja dois mínimos globais.

- A variável $passos$ é a quantidade de passos que foram necessários para se atingir o ótimo global.

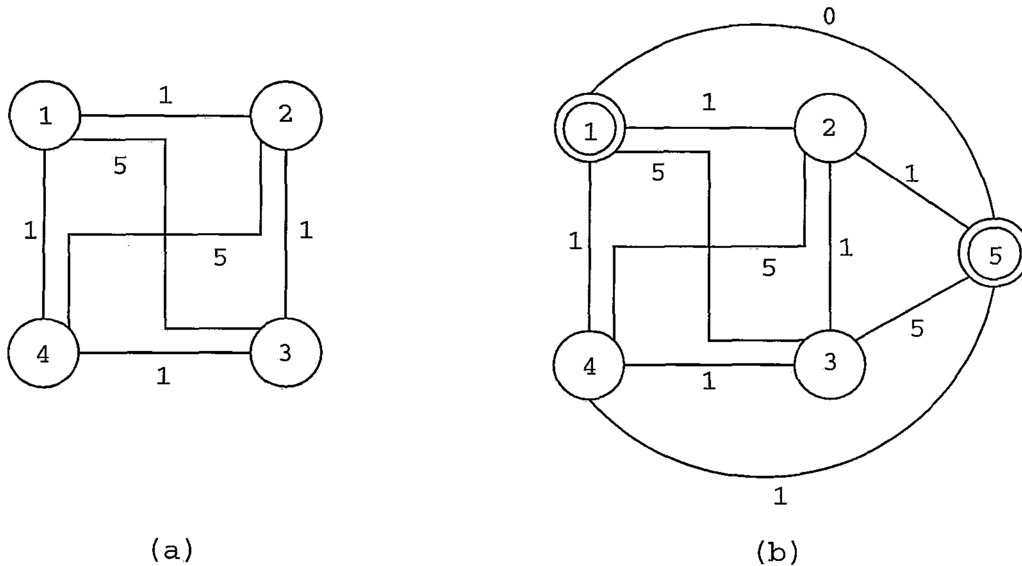


Figura 3.3.6: (a) Mapa Original das Cidades. (b) Mapa usado como Entrada para o SATyrus.

- A variável E é o valor da energia na última iteração.
- A variável T é o último valor da temperatura.
- Por último, se mostra o valor da melhor energia encontrada, ou seja, a energia do mínimo global.

Na Figura 3.3.8 (a) temos gerado o gráfico da primeira solução que o simulador encontrou e na Figura 3.3.8 (b) a segunda.

Finalmente, mostramos como é o comportamento da função de energia ao longo das iterações na Figura 3.3.9.

3.3.2 Simulando o Problema de Coloração de Grafos

Para ver como funciona o simulador no problema de coloração de grafos usaremos o grafo da Figura 3.3.10(a). Segundo o objetivo do problema cada um dos nós do grafo deve ser colorido com uma só cor, e a quantidade de cores utilizadas deve ser a menor possível. Considerando a restrição de que dois nós vizinhos devem ter cores diferentes o resultado do simulador pode ser visto na Figura 3.3.10(b).

Como no TSP, a saída do simulador dá também duas soluções ótimas. Estas podem ser lidas da matriz vc_{ij}. Recordemos que i representa o vértice e k o índice

```
mme@jaca:~/compiler4
Arquivo  Editar  Ver  Terminal  Abas  Ajuda
passos = 2199  ;  E = 14840  ;  T = 1.04495  ;  O estado da rede nao muda ha 1
passos = 2200  ;  E = 14840  ;  T = 1.04495  ;  O estado da rede nao muda ha 1
Estado da Rede (Solucao):
1-pos[1][1]-1  ;  1
2-pos[2][1]-0  ;  0
3-pos[3][1]-0  ;  0
4-pos[4][1]-0  ;  0
5-pos[5][1]-0  ;  0
6-pos[1][2]-0  ;  0
7-pos[2][2]-1  ;  0
8-pos[3][2]-0  ;  0
9-pos[4][2]-0  ;  1
10-pos[5][2]-0  ;  0
11-pos[1][3]-0  ;  0
12-pos[2][3]-0  ;  0
13-pos[3][3]-1  ;  1
14-pos[4][3]-0  ;  0
15-pos[5][3]-0  ;  0
16-pos[1][4]-0  ;  0
17-pos[2][4]-0  ;  1
18-pos[3][4]-0  ;  0
19-pos[4][4]-1  ;  0
20-pos[5][4]-0  ;  0
21-pos[1][5]-0  ;  0
22-pos[2][5]-0  ;  0
23-pos[3][5]-0  ;  0
24-pos[4][5]-0  ;  0
25-pos[5][5]-1  ;  1
Melhor Energia = 14840
Deseja continuar o annealing (Y ou N) ?
```

Figura 3.3.7: Saída do Simulador para o TSP.

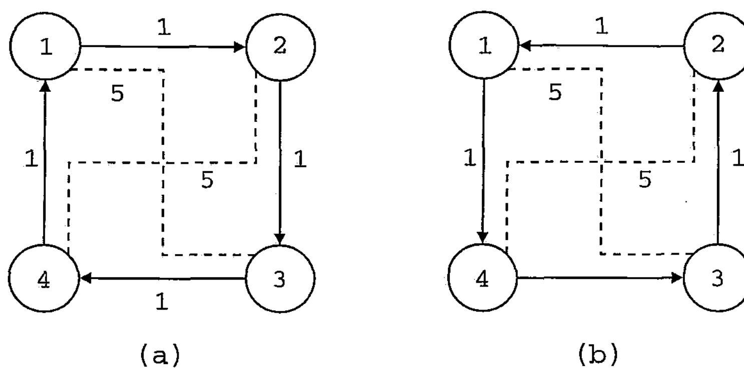


Figura 3.3.8: (a) Primeiro Caminho Encontrado pelo Simulador. (b) Segundo Caminho Encontrado pelo Simulador.

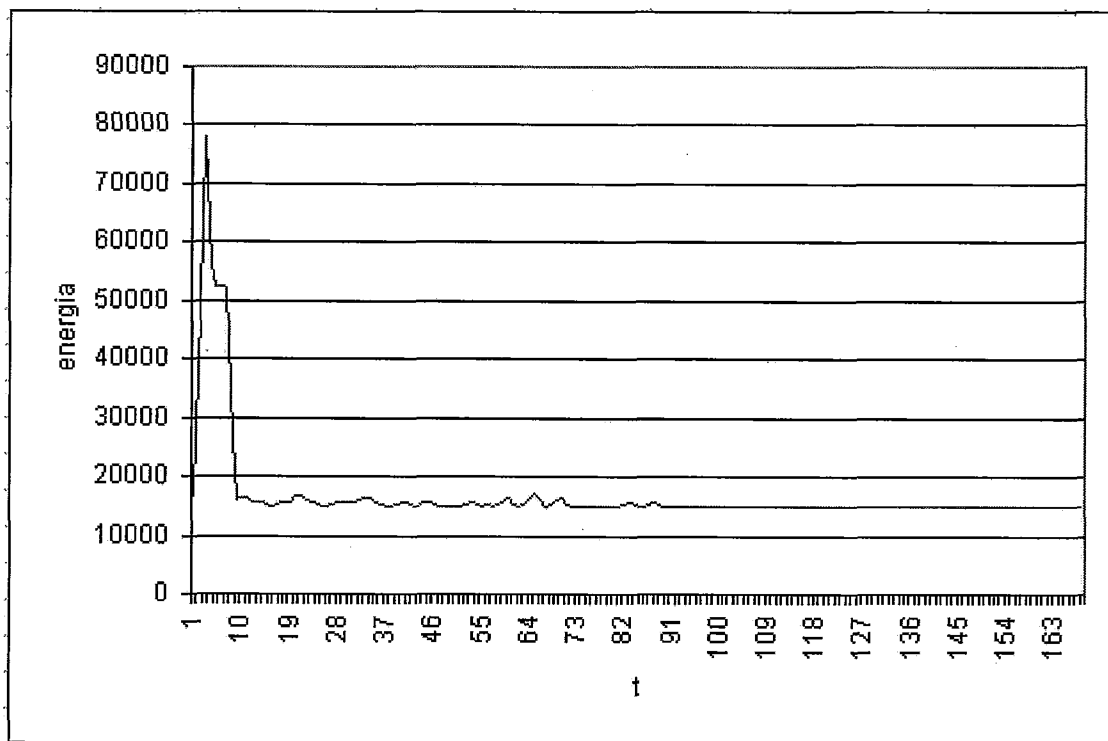


Figura 3.3.9: Trajetoria na Função de Energia do TSP.

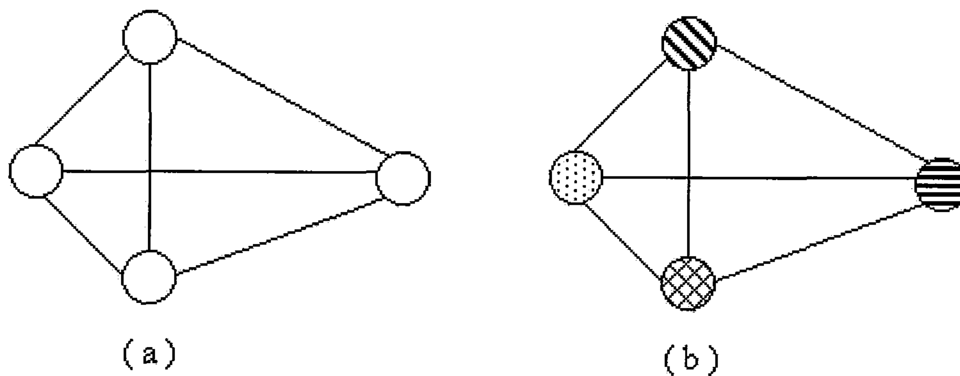


Figura 3.3.10: (a) Grafo Inicial a ser Colorido. (b) Resultado da Coloração do Grafo depois de rodar o Simulador.

```
karl@localhost:~/malela/compilador2
Arquivo  Editar  Ver  Terminal  Abas  Ajuda
passos = 3168  ;  E = 76  ;  T = 1.04495  ;  O estado da rede nao muda ha 73 ; 73 passos.
Estado da Rede (Solucao):
1-neigh[1][1]-0   ;  0
2-neigh[2][1]-1   ;  1
3-neigh[3][1]-1   ;  1
4-neigh[4][1]-1   ;  1
5-neigh[1][2]-1   ;  1
6-neigh[2][2]-0   ;  0
7-neigh[3][2]-1   ;  1
8-neigh[4][2]-1   ;  1
9-neigh[1][3]-1   ;  1
10-neigh[2][3]-1   ;  1
11-neigh[3][3]-0   ;  0
12-neigh[4][3]-1   ;  1
13-neigh[1][4]-1   ;  1
14-neigh[2][4]-1   ;  1
15-neigh[3][4]-1   ;  1
16-neigh[4][4]-0   ;  0
17-vc[1][1]-0   ;  1
18-vc[2][1]-1   ;  0
19-vc[3][1]-0   ;  0
20-vc[4][1]-0   ;  0
21-vc[1][2]-0   ;  0
22-vc[2][2]-0   ;  0
23-vc[3][2]-1   ;  1
24-vc[4][2]-0   ;  0
25-vc[1][3]-0   ;  0
26-vc[2][3]-0   ;  0
27-vc[3][3]-0   ;  0
28-vc[4][3]-1   ;  1
29-vc[1][4]-1   ;  0
30-vc[2][4]-0   ;  1
31-vc[3][4]-0   ;  0
32-vc[4][4]-0   ;  0
33-col[1]-1   ;  1
34-col[2]-1   ;  1
35-col[3]-1   ;  1
36-col[4]-1   ;  1
Melhor Energia = 76
Deseja continuar o annealing (Y ou N) ?
```

Figura 3.3.11: Saída do Simulador para o Problema de Coloração de Grafos.

da cor atribuída a este vértice. E a quantidade de cores utilizadas é igual ao número de neurônios ativados na matriz col_i. A Figura 3.3.11 mostra a saída gerada pelo simulador e finalmente na Figura 3.3.12 se observa o comportamento da função de energia até atingir o mínimo global.

Podemos encontrar mais exemplos sobre o funcionamento do simulador com outros problemas de maior complexidade em [24].

3.4 Comentários

Neste capítulo foram introduzidos os principais conceitos da arquitetura SATyrus. O simulador, um dos seus principais módulos, foi descrito com maior detalhe na Seção 3.2. Nesta se mostrou como a rede de Hopfield de alta-ordem é armazenada e como a atualização dos neurônios é feita.

No seguinte capítulo todos os detalhes sobre a construção e o funcionamento do compilador do SATyrus serão apresentados.

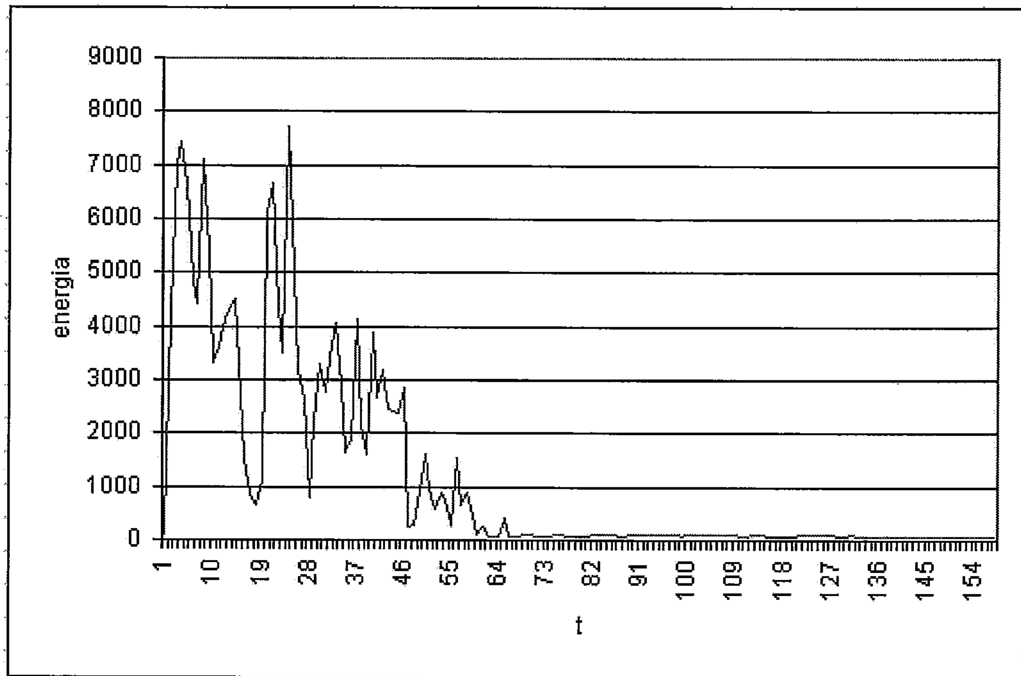


Figura 3.3.12: Trajetoria na Função de Energia do Problema de Coloração de Grafos.

Capítulo 4

O Compilador de Energia

Neste capítulo define-se e descreve-se o compilador de energia do SATyrus. Na Seção 4.1 detalha-se a linguagem do compilador. Na Seção 4.2 se dá uma visão geral da sua arquitetura. Nas seções posteriores se darão mais detalhes das fases de construção do compilador, tais como: análise léxica, análise sintática, geração da representação intermediária e a geração da função de energia.

4.1 Linguagem do SATyrus

A plataforma SATyrus provê ao usuário uma linguagem para a especificação de problemas tais como TSP e coloração de grafos, que terão sua equação de energia correspondente gerada. A linguagem é facilmente legível, inclusive para as pessoas que não a conhecem.

Os identificadores e as palavras-chave podem ser escritos em maiúsculas e minúsculas indistintamente, ou seja, é uma linguagem *case-insensitive*. Na Subseção 4.1.4 são apresentados dois exemplos usando a linguagem do SATyrus. Utilizamos a representação em negrito das palavras-chave para distingui-las das demais no código fonte.

Para comentar as linhas de código podem-se usar comentários de uma linha só ou múltiplas linhas. Para uma linha usa-se (//) e para múltiplas linhas (/*) no começo do comentário e (*/) no fim deste.

Um programa escrito na linguagem do SATyrus tem três partes obrigatórias:

(i) Definição das estruturas (que determinarão a geração dos neurônios);

(ii) Definição das restrições (que determinarão as vizinhanças dos neurônios);

(iii) Definição das penalidades (que determinarão a ordem de prioridade dos conjuntos de restrições).

4.1.1 Definição das Estruturas

Uma estrutura dentro do contexto da linguagem do SATyrus pode ser definida como:

(i) A atribução de um valor a um identificador;

(ii) Uma matriz de n dimensões onde cada elemento é um neurônio;

(iii) Uma matriz de n dimensões onde cada elemento é um valor numérico que serve de coeficiente para algumas das parcelas da parte da equação de energia relativa às restrições de otimalidade.

As estruturas são separadas por ponto e vírgula (;) e devem ser obrigatoriamente declaradas antes de se começar a definição das restrições.

4.1.1.1 Atribuições a um Identificador

Chama-se identificador a uma cadeia alfa-numérica que deve começar sempre com uma letra e ser diferente das palavras-chave. As possíveis atribuições são:

(i) Um valor constante.

(ii) A soma ou subtração de um identificador com um valor constante, ou vice-versa.

(iii) A soma ou subtração entre dois identificadores.

Exemplos:

- `num=4;`
- `num1=num+2;`
- `num2=num1-num;`

4.1.1.2 Matrizes de Neurônios

A declaração de uma matriz de neurônios tem o seguinte formato:

$$nome_da_matriz\ (d_1, d_2, ..., d_n)$$

onde:

- *nome_ da_ matriz* é uma cadeia alfa-numérica que deve começar sempre com uma letra.

- $d_1, d_2, ..., d_n$ são as dimensões da matriz, que podem ser definidas utilizando: números, identificadores, intervalos ou a combinação de quaisquer deles. Para separar as dimensões, cujo número é irrestrito, utilizamos vírgulas.

Vejamos exemplos de como definir uma matriz:

(i) Usando números:

Exemplos:

- `matriz1(4);`
- `matriz2(5,2);`

(ii) Usando identificadores, os quais devem ser previamente definidos. Também pode-se usar soma ou substração de um identificador com um número, ou vice-versa; ou entre dois identificadores.

Exemplo:

```
num=4;
num1=2;
matriz1(num);
matriz2(num+1,num-num1);
```

(iii) Usando intervalos, os quais têm o seguinte formato geral:

lim_ inf sinal_ desigualdade nome_ índice sinal_ desigualdade lim_ sup

Onde:

- O *lim_inf* e *lim_sup* podem ser um número, um identificador ou outro índice declarado previamente como uma das dimensões da matriz que se está definindo. Adicionalmente, pode-se realizar somas ou subtrações entre eles.
- O *sinal_desigualdade* é < ou <=.
- O *nome_índice* é uma letra só. O índice é único para cada intervalo, isto é, cada um dos intervalos usados para definir as dimensões de uma mesma matriz devem ter índices diferentes.

Exemplos:

- matriz1(1 <= i <= 5,1 <= j <= 2);
- num=5;
  num1=2;
  matriz1(1 <= i <= num,1 <= j <= num1);
- num=1;
  num1=7;
  matriz1(num <= i <= num1-num2,num <= j <= num+1);
- num=4;
  matriz3(1 <= i <= num, i+1 <= j <= num);

Existem casos nos quais se deseja aplicar restrições de desigualdade sobre os índices, ou seja, evitar que se crie um neurônio com dois índices, ou mais, com o mesmo valor. Neste caso, as diferenças são escritas depois da definição de todos os intervalos. Estas são separadas dos intervalos por ponto e vírgula (;) e entre elas por vírgula (,).

Exemplos:

- matriz4(1 <= i <= 5, 1 <= j <= 5; j ! = i);
- matriz5(1 <= i <= 5, 1 <= j <= 5, 1 <= k <= 2, 1 <= l <= 2; j ! = i, k ! = l);

4.1.1.3 Matrizes de Valores Numéricos

São declaradas do mesmo jeito que as matrizes de neurônios. A única diferença é que deve-se especificar o nome do arquivo de texto (.txt) do qual serão lidos os valores.

Exemplo:

- dist (num,num);
  dist **read from** tsp1.txt;

4.1.2 Definição das Restrições

Na primeira parte do formato usado para declarar cada uma das restrições tem-se que definir o grupo, a penalidade e os índices que se usarão nas cláusulas.

grupo_da_restrição **group type** *nome_da_penalidade* : **forall** { *nomes_dos_índices* };

- Existem dois grupos de restrições: de integridade (*integrity*) e de otimalidade (*optimality*). Uma restrição pertence somente a um desses dois.
- O *nome_da_penalidade*, o qual identifica a penalidade de um grupo de cláusulas, é uma cadeia alfa-numérica que começa sempre com uma letra. Os nomes das penalidades são associados a níveis de penalidades ao final do código fonte. Cada nível de penalidade pode ter associado mais de um *nome_da_penalidade*.
- Os *nomes_dos_índices*, como nos intervalos, são uma letra só. Para separá-los usa-se vírgulas (,). Deve-se declarar obrigatoriamente todos os índices que serão usados nas cláusulas.

Na segunda parte do formato tem-se a definição dos limites dos índices, das desigualdades e das atribuições de valores aos índices:

definições_dos_limites [; *desigualdades_e_atribuições*] :

- As *definições_dos_limites* seguem o formato explicado para os intervalos na Sub-seção 4.1.1.2. Estas definições são obrigatórias.

- As *desigualdades_ e_ atribuições* são opcionais. A definição de desigualdades é feita do mesmo jeito que a dos intervalos. As atribuições são usadas quando alguns dos limites tem um valor só, então não se necessita usar um intervalo. O valor atribuído pode ser um inteiro, um identificador ou a soma ou subtração destes.

Na última parte são definidas as cláusulas na Forma Normal Conjuntiva [28], ou seja, como uma conjunção de disjunções. Quando a cláusula é um só literal não se precisa de parênteses, do contrário se terá parênteses para cada disjunto. Uma informação adicional à respeito dos índices das cláusulas é que podem ser feitas somas ou subtrações dos índices com um inteiro ou com um identificador. O formato geral de uma cláusula de mais de um literal é:

$clausula_i$ = $(disjunto_1)$ **and** $(disjunto_2)$ **and** ... $(disjunto_n)$
$(disjunto_j)$ = $(literal_1$ **or** $literal_2$ **or** ... $literal_m)$
$literal_k$= $atomo_l$ ou $literal_k$=**not** $atomo_l$

Nas restrições de otimalidade, existem casos que para representar a função objetivo é necessário utilizar coeficientes numéricos para algumas das parcelas. Nestes casos, o formato da cláusula varia a:

$clausula_i$ = $coeficiente(disjuntos)$;
$disjuntos$ = $(disjunto_1)$ **and** $(disjunto_2)$ **and** ... $(disjunto_n)$
$(disjunto_j)$ = $(literal_1$ **or** $literal_2$ **or** ... $literal_m)$
$literal_k$= $atomo_l$ ou $literal_k$=**not** $atomo_l$

As restrições, conforme as estruturas, são separadas por ponto e vírgula (;).

4.1.3 Definição das Penalidades

O formato para definir as penalidades é o seguinte:

penalty { *definições_ dos_ níveis_ das_ penalidades* }

Nas *definições_ dos_ níveis_ das_ penalidades* se dá a cada penalidade o valor do seu nível de prioridade. Começa-se sempre com 0, que é o nível mais baixo. Para definir o nível de cada penalidade usa-se o formato:

nome_ da_ penalidade **is level** *valor_ inteiro*;

4.1.4 Exemplos

A seguir os dois exemplos, já modelados anteriormente em [21], escritos na linguagem do compilador do SATyrus.

4.1.4.1 Código de TSP

```
//DEFINIÇÃO DAS ESTRUTURAS
num=5; /* cinco cidades serão consideradas*/
pos(num,num);
dist (num,num);
dist read from tsp1.txt; /* tsp1.txt contém as distâncias entre cidades*/
//DEFINIÇÃO DAS RESTRIÇÕES
integrity group type int1: forall {i,j}; 1<=i<=num,1<=j<=num: pos[i][j];
integrity group type wta: forall {i,j,k}; 1<=i<=num,1<=j<=num,1<=k<=num;i!=k: (not
pos[i][j] or not pos[k][j]);
integrity group type wta: forall {i,j,l}; 1<=i<=num,1<=j<=num,1<=l<=num;j!=l: (not
pos[i][j] or not pos[i][l]);
optimality group type custo: forall {i,j,k}; 1<=i<=num,1<=j<=4,1<=k<=num;i!=k:
dist[i][k](pos[i][j] and pos[k][j+1]);
optimality group type custo: forall {i,j,k}; 1<=i<=num,2<=j<=num,1<=k<=num;i!=k:
dist[i][k](pos[i][j] and pos[k][j-1]);
//DEFINIÇÃO DAS PENALIDADES
penalty{
wta is level 2;
int1 is level 1;
custo is level 0;}
```

4.1.4.2 Código do Problema de Coloração de Grafos

```
//DEFINIÇÃO DAS ESTRUTURAS
num=4; /*um grafo com 4 nós será colorido*/
neigh(num,num);
vc(num,num);
col(num);
```

```
//DEFINIÇÃO DAS RESTRIÇÕES
integrity group type int1: forall{i,k}; 1<=i<=num,1<=k<=num: vc[i][k];
integrity group type int1: forall{i,l,k}; 1<=i<=num,1<=l<=num,1<=k<=num;i!=l: (not neigh[i][l] or not vc[i][k] or not vc[l][k]);
integrity group type wta: forall{i,k,m}; 1<=i<=num,1<=k<=num,1<=m<=num;k!=m: (not vc[i][k] or not vc[i][m]);
integrity group type int1: forall{i,k}; 1<=i<=num,1<=k<=num: (not vc[i][k] or col[k]);
optimality group type custo: forall{k}; 1<=k<=num: col[k];
//DEFINIÇÃO DAS PENALIDADES
penalty{
wta is level 2;
int1 is level 1;
custo is level 0;}
```

4.2 Visão Geral do Compilador

Para se transformar as restrições em uma Função de Energia, a função H^* é usada. Quando os problemas são pequenos é possível fazer estas operações manualmente. Não obstante, quanto maior o número de cálculos, mais complexos estes se tornam e maior a possibilidade de haver erros. A solução é fazer com que todos os cálculos necessários para gerar a função de energia sejam automatizados. O encarregado de realizar este processo e gerar a Função de Energia é o compilador do SATyrus. Sua função é transformar o conjunto de restrições escritas na linguagem do SATyrus (código fonte) na sua Função de Energia equivalente (código objeto).

Na Figura 4.2.1 se mostra a arquitetura do compilador. As entradas são as restrições do problema escritas na linguagem do SATyrus e os valores iniciais dos neurônios. E a saída é a função de energia. O detalhe de cada um dos módulos será visto nas seções seguintes.

4.3 Análise Léxica

A análise léxica é a fase onde o código fonte é lido caracter por caracter. Os caracteres são agrupados em cadeias que representam componentes léxicos chamados

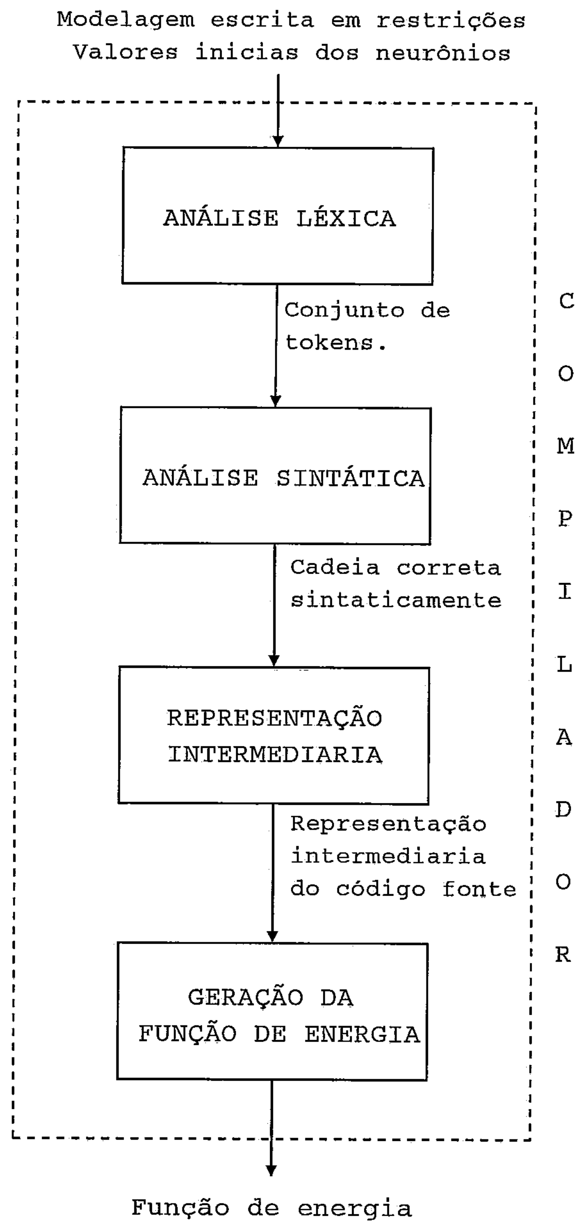


Figura 4.2.1: Arquitetura do Compilador.

tokens, os quais representam os nomes dos identificadores, operadores, sinais de pontuação, palavras-chave e tudo o que o código fonte possa conter. Os *tokens* gerados são usados pelo analisador sintático para determinar se o código fonte está gramaticalmente correto. Os comentários e os espaços em branco são descartados e não são considerados na análise sintática.

A Tabela 4.1 mostra os *tokens* que formam o vocabulário que é utilizado pelo analisador léxico deste compilador e suas respectivas descrições.

Para gerar o analisador léxico utilizamos Flex++ [11]. Este lê um arquivo de entrada que contém as especificações sobre os *tokens* e outras características que se deseje dar ao analisador léxico. Este arquivo é compilado com as próprias bibliotecas do Flex++ para produzir o código correspondente em C++. O código do analisador léxico em Flex++ pode ser encontrado no Apêndice B.

4.4 Análise Sintática

O analisador sintático, também chamado *parser* [1], recebe como entrada os *tokens* enviados pelo analisador léxico e verifica se estes estão na ordem permitida pela gramática da linguagem. A gramática da linguagem do SATyrus pode ser encontrada no Apêndice C.

A ferramenta utilizada para gerar o analisador sintático foi o Bison++ [10]. A entrada para o Bison++ é um arquivo que contém a gramática da linguagem, que deve ser livre de contexto. Esta gramática pode ser definida mediante a quádrupla: $G = (V_t, V_n, P, S)$.

Onde:

V_t é o conjunto finito de terminais.

V_n é o conjunto finito de não terminais.

P é o conjunto finito de produções.

$S \in V_n$ é denominado símbolo inicial.

Os elementos de P são da forma:

$V_n - > (V_t \cup V_n)^*$

A gramática de entrada para o Bison++ está escrita na *Backus-Naur Form* (BNF) (Naur [23] *apud* Russel [28]). Os símbolos na BNF utilizados para definir

Token	Descrição
integrity	Palavra chave usada para definir uma restrição de integridade.
optimality	Palavra chave usada para definir uma restrição de otimalidade.
group	Palavra chave usada para definir o grupo de uma restrição.
forall	Palavra chave usada para definir índices.
type	Palavra chave usada para definir o tipo de penalidade.
penalty	Palavra chave usada para definir os níveis das penalidades.
level	Palavra chave usada para definir o nível da penalidade.
is	Palavra chave usada para definir o nível da penalidade.
read	Palavra chave usada para definir o nome do arquivo do qual serão lidos os valores.
from	Palavra chave usada para definir o nome do arquivo do qual serão lidos os valores.
or	Operador lógico OR.
and	Operador lógico AND.
not	Operador lógico NOT.
equal	Operador aritmético $=$.
less	Operador aritmético $<$.
noteq	Operador aritmético $! =$.
plus	Operador aritmético $+$.
minus	Operador aritmético $-$.
comma	Sinal de pontuação ",".
semicolon	Sinal de pontuação ";".
colon	Sinal de pontuação ":".
rparent	Sinal de pontuação ")".
lparent	Sinal de pontuação "(".
rcbracket	Sinal de pontuação "}".
lcbracket	Sinal de pontuação "{".
rbracket	Sinal de pontuação "]".
lbracket	Sinal de pontuação "[".
file	Palavra chave que se usa para definir o nome de um arquivo de valores.
varprop	Um *varprop* é uma cadeia alfa-numérica que começa sempre com uma letra.
integer	Um *integer* é uma seqüência de dígitos que definem um número inteiro.
float	Um *float* é uma seqüência de dígitos que definem um número flutuante.
index	Um *index* é uma letra só que define um índice.

Tabela 4.1: Tabela de *tokens* e suas Respectivas Descrições.

uma gramática são:

::=	significa "é definido como".
|	representa o OR.
<>	para encerrar símbolos não terminais.

O Bison++ trabalha com gramáticas do tipo *Look-Ahead LR* (LALR(1)) [1]. Estas são do tipo *bottom-up* e lêem a entrada da esquerda (L_{eft}) para a direita (R_{ight}) e durante a análise lêem 1 *token* com antecipação. É preferível que a gramática da linguagem não seja ambígua para evitar conflitos nas ações de redução.

4.5 Geração da Representação Intermediária

Depois de saber que o código fonte é correto léxica e sintaticamente, o passo seguinte é criar uma representação intermediária explícita deste. A função desta representação é ajudar na geração futura do código objeto, em nosso caso a Função de Energia.

Para gerar a Função de Energia é preciso traduzir o código fonte para uma estrutura de dados auxiliar ao processo de geração. Esta estrutura é mostrada na Figura 4.5.2. Uma descrição de cada parte da arquitetura é feita a seguir:

- A **lista de cláusulas** guarda o conjunto de cláusulas definidas para descrever as restrições.
- A **cláusula** é escrita usando-se uma ou mais estruturas. Toda cláusula está associada a um grupo e um nível de penalidade armazenado em uma lista de penalidades. Cada cláusula possui também um conjunto de índices associado.
- A **lista de estruturas** guarda todas as estruturas declaradas.
- A **estrutura** pode ser uma matriz, como já foi dito. Neste caso é necessário armazenar o número de dimensões e o valor de cada uma. Em caso contrário, se guarda só o valor que lhe foi atribuído.
- A **lista de penalidades** guarda todos os níveis de penalidades usados.
- A **penalidade** tem um identificador, valor e nível de prioridade associado.

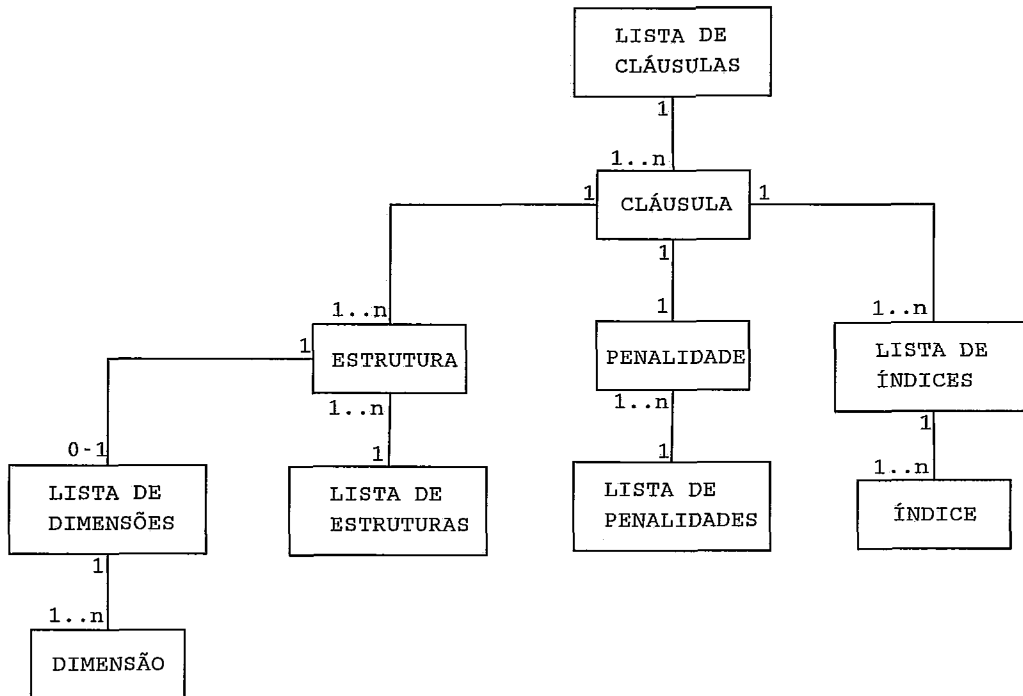


Figura 4.5.2: Diagrama de Classes da Representação Intermediária.

- A **lista de índices** guarda os índices que serão usados na definição da cláusula. Tem-se uma lista diferente para cada cláusula definida.

- O **índice** tem um identificador e limites superior e inferior. Além disto pode ter restrições de desigualdade com outros índices.

- A **lista de dimensões** guarda as dimensões de uma matriz.

- A **dimensão** tem um identificador e limites superior e inferior.

O gerador da representação intermediária, se encarrega de conferir e assegurar que:

- Cada uma das penalidades usadas nas restrições tenha sido armazenada na lista de penalidades.

- Cada um dos índices utilizados para a definição de uma cláusula tenha sido armazenado na lista de índices da dita cláusula.

- Não existam índices declarados e não usados.

- Cada uma das estruturas utilizadas para definir uma cláusula tenha sido declarada e armazenada na lista de estruturas.
- Não se usem estruturas não declaradas previamente.
- O número de dimensões definido para uma matriz (estrutura) seja respeitado ao se usar a dita matriz nas cláusulas.

4.6 Geração da Função de Energia

Esta constitui o passo final do compilador. Nesta fase vai-se gerar o código objeto (Função de Energia) a partir da informação armazenada. Os passos deste processo são:

(i) Criação dos neurônios;

(ii) Geração das parcelas.

4.6.1 Criação dos Neurônios

Os neurônios são criados a partir das matrizes declaradas. A quantidade de neurônios depende das dimensões destas e das restrições nos seus limites inferior e superior. Por exemplo:

- Da matriz `pos(3,3)` criam-se 9 neurônios.
- Da matriz `val(1<=i<=4,1<=j<=4;i!=j)` criam-se 12 neurônios.
- Da matriz `neigh(1<=i<=4,i+1<=j<=4)` criam-se 6 neurônios.

Cada neurônio criado tem associado um identificador e um nome. Este está composto pelo nome da matriz, da qual o neurônio faz parte, mais as dimensões desta entre colchetes. A Tabela 4.2 ilustra a atribuição de nomes e *IDs* aos neurônios representados pelos possíveis elementos de uma matriz 3×3 chamada *pos*.

Os neurônios com seus atributos são armazenados numa lista de neurônios, para sua posterior utilização. Seus atributos, além do seu nome, são seu estado, sua probabilidade e um atributo chamado fixo (para indicar se o valor do estado

Dim1	Dim2	Nome do neurônio	ID
1	1	pos[1][1]	1
1	2	pos[1][2]	2
1	3	pos[1][3]	3
2	1	pos[2][1]	4
2	2	pos[2][2]	5
2	3	pos[2][3]	6
3	1	pos[3][1]	7
3	2	pos[3][2]	8
3	3	pos[3][3]	9

Tabela 4.2: Tabela com todos os Identificadores e Nomes dos Neurônios da Matriz $pos(3,3)$.

dado para esse neurônio pode ou não ser mudado). A probabilidade é sempre inicializada com 1. Para o caso dos estados, estes podem ser inicializados:

- Com 0;
- Com 1;
- Aleatoriamente;

Adicionalmente, depois deste procedimento geral, podem-se inicializar alguns ou todos os neurônios com o estado que se deseje. Para isto é necessário criar um arquivo com o nome do neurônio. Cada linha do arquivo inicializa um neurônio, o que é representado pelas suas coordenadas separadas por vírgulas. Após as coordenadas, seguem o valor do estado(0/1), o atributo fixo (0/1) e a probabilidade, separados por hífen. Por exemplo, depois de se ter inicializado os neurônios da matriz *pos* aleatoriamente, se deseja que o estado do neurônio $pos[2][3]$ seja exatamente 1. Então cria-se o arquivo de nome *pos* com uma linha só para inicializá-lo. O conteúdo da linha é: $2,3-1-1-1$. Onde $2,3$ é a coordenada do neurônio, o primeiro 1 é o valor atribuído ao estado, o segundo 1 diz que o valor do estado não deve mudar e último 1 a probabilidade.

Se os neurônios tivessem coeficientes existiria uma lista onde cada coeficiente tem um identificador, que seria gerado do mesmo jeito que para os neurônios, e um valor atribuído.

ID da penalidade	Valor do peso	Valor da aridade	Lista dos IDs dos neurônios

Figura 4.6.3: Informação Associada a cada Parcela da Função de Energia.

4.6.2 Geração das Parcelas

Para começar a gerar as parcelas da função de energia se aplica a cada uma das cláusulas a função H^*. Por exemplo, seja:

integrity group type penal1: **forall**{i,j}; 1<=i<=3, 1<=j<=3: not res[i][j] or not iin[i][j];

o conjunto de cláusulas das quais se deseja gerar a Função de Energia.

Depois de se aplicar a função H^* na cláusula, esta se transforma em: $\sum_{i=1}^{3}\sum_{j=1}^{3}(res[i][j]iin[i][j])$, que será como um padrão da função, a partir do qual serão geradas todas as parcelas finais.

O gerador da Função de Energia tem funções que realizam estes cálculos automatizadamente e que deixam o padrão da função pronto. Isto significa que nenhum cálculo de multiplicação complexo precisa ser feito.

Agora pode-se gerar cada uma das parcelas deste somatório. Os dados de cada parcela serão armazenados numa estrutura como a mostrada na Figura 4.6.3. A seguir descreveremos cada um deles:

- O ID da penalidade, corresponde ao índice desta na lista de penalidades.
- O valor do peso, é um dado acumulativo que vai somando ou subtraindo, conforme o sinal, o valor da parcela quando esta se repete mais de uma vez na função.
- A aridade é o dado que contém o número de neurônios envolvidos na conexão.
- A lista de IDs dos neurônios, é um vetor que contém o índice de cada um dos neurônios que são parte da conexão. O índice é a posição do neurônio na lista de neurônios.

No passo seguinte, para cada grupo são geradas todas as cláusulas através das combinações possíveis dos índices. Deve-se levar em conta as possíveis restrições de diferença entre índices ou índices com um valor só. Na Tabela 4.3 estão todas as

i	j	ID Pen.	Peso	Aridade	Neurônios
1	1	0	1	2	res[1][1] iin[1][1]
1	2	0	1	2	res[1][2] iin[1][2]
1	3	0	1	2	res[1][3] iin[1][3]
2	1	0	1	2	res[2][1] iin[2][1]
2	2	0	1	2	res[2][2] iin[2][2]
2	3	0	1	2	res[2][3] iin[2][3]
3	1	0	1	2	res[3][1] iin[3][1]
3	2	0	1	2	res[3][2] iin[3][2]
3	3	0	1	2	res[3][3] iin[1][3]

Tabela 4.3: Tabela das Parcelas Geradas a partir do Padrão $\sum_{i=1}^{3}\sum_{j=1}^{3}(res[i][j]iin[i][j])$.

possíveis combinações para os índices da cláusula do exemplo anterior junto com as parcelas resultantes. Neste caso não se tem os mesmos neurônios mais de uma vez, nem coeficientes, desta forma o peso fica em 1. A respeito do ID da penalidade, supondo que a penalidade armazenada é uma só, então seu ID é 0. Por último, a aridade em todos os casos é 2 porque os neurônios envolvidos na conexão são dois.

Durante o processo de geração das parcelas, vai-se contando o número destas em cada nível de penalidade. Este valor serve para fazer o cálculo final do valor de cada penalidade. Este pode ser alterado posteriormente para ajudar no processo de convergência.

4.7 Exemplos de Compilação

No capítulo anterior vimos o resultado da simulação do TSP e do problema de coloração de grafos. Nesta seção se mostrarão as saídas respectivas da compilação, utilizadas como entradas do simulador.

4.7.1 Compilando o TSP

As duas saídas principais do compilador são as parcelas da Função de Energia e os neurônios com seus respectivos atributos. Na Figura 4.7.4 se mostra a primeira parte das parcelas geradas para o TSP e na Figura 4.7.5 os neurônios.

```
mme@mamao:~/compiler4
Arquivo  Editar  Ver  Terminal  Abas  Ajuda
PARCELAS
id: 1;  penal: 1, peso: 25, arid: 0; Neuronios:
id: 2;  penal: 1, peso: -1, arid: 1; Neuronios: pos[1][1],
id: 3;  penal: 1, peso: -1, arid: 1; Neuronios: pos[2][1],
id: 4;  penal: 1, peso: -1, arid: 1; Neuronios: pos[3][1],
id: 5;  penal: 1, peso: -1, arid: 1; Neuronios: pos[4][1],
id: 6;  penal: 1, peso: -1, arid: 1; Neuronios: pos[5][1],
id: 7;  penal: 1, peso: -1, arid: 1; Neuronios: pos[1][2],
id: 8;  penal: 1, peso: -1, arid: 1; Neuronios: pos[2][2],
id: 9;  penal: 1, peso: -1, arid: 1; Neuronios: pos[3][2],
id: 10;  penal: 1, peso: -1, arid: 1; Neuronios: pos[4][2],
id: 11;  penal: 1, peso: -1, arid: 1; Neuronios: pos[5][2],
id: 12;  penal: 1, peso: -1, arid: 1; Neuronios: pos[1][3],
id: 13;  penal: 1, peso: -1, arid: 1; Neuronios: pos[2][3],
id: 14;  penal: 1, peso: -1, arid: 1; Neuronios: pos[3][3],
id: 15;  penal: 1, peso: -1, arid: 1; Neuronios: pos[4][3],
id: 16;  penal: 1, peso: -1, arid: 1; Neuronios: pos[5][3],
id: 17;  penal: 1, peso: -1, arid: 1; Neuronios: pos[1][4],
id: 18;  penal: 1, peso: -1, arid: 1; Neuronios: pos[2][4],
id: 19;  penal: 1, peso: -1, arid: 1; Neuronios: pos[3][4],
id: 20;  penal: 1, peso: -1, arid: 1; Neuronios: pos[4][4],
id: 21;  penal: 1, peso: -1, arid: 1; Neuronios: pos[5][4],
id: 22;  penal: 1, peso: -1, arid: 1; Neuronios: pos[1][5],
id: 23;  penal: 1, peso: -1, arid: 1; Neuronios: pos[2][5],
id: 24;  penal: 1, peso: -1, arid: 1; Neuronios: pos[3][5],
id: 25;  penal: 1, peso: -1, arid: 1; Neuronios: pos[4][5],
id: 26;  penal: 1, peso: -1, arid: 1; Neuronios: pos[5][5],
id: 27;  penal: 2, peso: 2, arid: 2; Neuronios: pos[2][1],  pos[1][1],
id: 28;  penal: 2, peso: 2, arid: 2; Neuronios: pos[3][1],  pos[1][1],
id: 29;  penal: 2, peso: 2, arid: 2; Neuronios: pos[4][1],  pos[1][1],
id: 30;  penal: 2, peso: 2, arid: 2; Neuronios: pos[5][1],  pos[1][1],
id: 31;  penal: 2, peso: 2, arid: 2; Neuronios: pos[2][2],  pos[1][2],
id: 32;  penal: 2, peso: 2, arid: 2; Neuronios: pos[3][2],  pos[1][2],
id: 33;  penal: 2, peso: 2, arid: 2; Neuronios: pos[4][2],  pos[1][2],
id: 34;  penal: 2, peso: 2, arid: 2; Neuronios: pos[5][2],  pos[1][2],
id: 35;  penal: 2, peso: 2, arid: 2; Neuronios: pos[2][3],  pos[1][3],
id: 36;  penal: 2, peso: 2, arid: 2; Neuronios: pos[3][3],  pos[1][3],
id: 37;  penal: 2, peso: 2, arid: 2; Neuronios: pos[4][3],  pos[1][3],
id: 38;  penal: 2, peso: 2, arid: 2; Neuronios: pos[5][3],  pos[1][3],
```

Figura 4.7.4: Parte das Parcelas da Função de Energia do TSP.

```
Arquivo  Editar  Ver  Terminal  Abas  Ajuda
NEURONIOS
num: 1    nome: pos[1][1]    val: 1    fixo: 1    prob: 1
num: 2    nome: pos[2][1]    val: 0    fixo: 0    prob: 1
num: 3    nome: pos[3][1]    val: 0    fixo: 0    prob: 1
num: 4    nome: pos[4][1]    val: 0    fixo: 0    prob: 1
num: 5    nome: pos[5][1]    val: 0    fixo: 0    prob: 1
num: 6    nome: pos[1][2]    val: 0    fixo: 0    prob: 1
num: 7    nome: pos[2][2]    val: 0    fixo: 0    prob: 1
num: 8    nome: pos[3][2]    val: 0    fixo: 0    prob: 1
num: 9    nome: pos[4][2]    val: 0    fixo: 0    prob: 1
num: 10    nome: pos[5][2]    val: 0    fixo: 0    prob: 1
num: 11    nome: pos[1][3]    val: 0    fixo: 0    prob: 1
num: 12    nome: pos[2][3]    val: 0    fixo: 0    prob: 1
num: 13    nome: pos[3][3]    val: 0    fixo: 0    prob: 1
num: 14    nome: pos[4][3]    val: 0    fixo: 0    prob: 1
num: 15    nome: pos[5][3]    val: 0    fixo: 0    prob: 1
num: 16    nome: pos[1][4]    val: 0    fixo: 0    prob: 1
num: 17    nome: pos[2][4]    val: 0    fixo: 0    prob: 1
num: 18    nome: pos[3][4]    val: 0    fixo: 0    prob: 1
num: 19    nome: pos[4][4]    val: 0    fixo: 0    prob: 1
num: 20    nome: pos[5][4]    val: 0    fixo: 0    prob: 1
num: 21    nome: pos[1][5]    val: 0    fixo: 0    prob: 1
num: 22    nome: pos[2][5]    val: 0    fixo: 0    prob: 1
num: 23    nome: pos[3][5]    val: 0    fixo: 0    prob: 1
num: 24    nome: pos[4][5]    val: 0    fixo: 0    prob: 1
num: 25    nome: pos[5][5]    val: 1    fixo: 1    prob: 1
```

Figura 4.7.5: Neurônios Gerados para uma Instância de TSP com n=5.

```
mme@mamao:~/compiler4
Arquivo  Editar  Ver  Terminal  Abas  Ajuda
PARCELAS
id: 1;   penal: 1, peso: 16, arid: 0; Neuronios:
id: 2;   penal: 1, peso: 0, arid: 1; Neuronios: vc[1][1],
id: 3;   penal: 1, peso: 0, arid: 1; Neuronios: vc[2][1],
id: 4;   penal: 1, peso: 0, arid: 1; Neuronios: vc[3][1],
id: 5;   penal: 1, peso: 0, arid: 1; Neuronios: vc[4][1],
id: 6;   penal: 1, peso: 0, arid: 1; Neuronios: vc[1][2],
id: 7;   penal: 1, peso: 0, arid: 1; Neuronios: vc[2][2],
id: 8;   penal: 1, peso: 0, arid: 1; Neuronios: vc[3][2],
id: 9;   penal: 1, peso: 0, arid: 1; Neuronios: vc[4][2],
id: 10;   penal: 1, peso: 0, arid: 1; Neuronios: vc[1][3],
id: 11;   penal: 1, peso: 0, arid: 1; Neuronios: vc[2][3],
id: 12;   penal: 1, peso: 0, arid: 1; Neuronios: vc[3][3],
id: 13;   penal: 1, peso: 0, arid: 1; Neuronios: vc[4][3],
id: 14;   penal: 1, peso: 0, arid: 1; Neuronios: vc[1][4],
id: 15;   penal: 1, peso: 0, arid: 1; Neuronios: vc[2][4],
id: 16;   penal: 1, peso: 0, arid: 1; Neuronios: vc[3][4],
id: 17;   penal: 1, peso: 0, arid: 1; Neuronios: vc[4][4],
id: 18;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[2][1],   vc[2][1],   vc[1][1],
id: 19;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[3][1],   vc[3][1],   vc[1][1],
id: 20;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[4][1],   vc[4][1],   vc[1][1],
id: 21;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[1][2],   vc[1][1],   vc[2][1],
id: 22;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[3][2],   vc[3][1],   vc[2][1],
id: 23;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[4][2],   vc[4][1],   vc[2][1],
id: 24;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[1][3],   vc[1][1],   vc[3][1],
id: 25;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[2][3],   vc[2][1],   vc[3][1],
id: 26;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[4][3],   vc[4][1],   vc[3][1],
id: 27;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[1][4],   vc[1][1],   vc[4][1],
id: 28;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[2][4],   vc[2][1],   vc[4][1],
id: 29;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[3][4],   vc[3][1],   vc[4][1],
id: 30;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[2][1],   vc[2][2],   vc[1][2],
id: 31;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[3][1],   vc[3][2],   vc[1][2],
id: 32;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[4][1],   vc[4][2],   vc[1][2],
id: 33;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[1][2],   vc[1][2],   vc[2][2],
id: 34;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[3][2],   vc[3][2],   vc[2][2],
id: 35;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[4][2],   vc[4][2],   vc[2][2],
id: 36;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[1][3],   vc[1][2],   vc[3][2],
id: 37;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[2][3],   vc[2][2],   vc[3][2],
id: 38;   penal: 1, peso: 1, arid: 3; Neuronios: neigh[4][3],   vc[4][2],   vc[3][2],
```

Figura 4.7.6: Partes das Parcelas da Função de Energia do Problema de Coloração de Grafos.

4.7.2 Compilando o Problema de Coloração de Grafos

Na Figura 4.7.6 se mostram as primeiras parcelas geradas para o problema de coloração de grafos e na Figura 4.7.7 os neurônios deste.

4.8 Comentários

Este capítulo mostra as características e funcionamento do compilador do SATyrus. A linguagem própria deste foi definida e descrita na Seção 4.1 e nas seções subseqüentes foram mostradas a arquitetura, as características e o funcionamento das

Arquivo Editar Ver Terminal Abas Ajuda

```
NEURONIOS
num: 1   nome: neigh[1][1]   val: 0   fixo: 1   prob: 1
num: 2   nome: neigh[2][1]   val: 1   fixo: 1   prob: 1
num: 3   nome: neigh[3][1]   val: 1   fixo: 1   prob: 1
num: 4   nome: neigh[4][1]   val: 1   fixo: 1   prob: 1
num: 5   nome: neigh[1][2]   val: 1   fixo: 1   prob: 1
num: 6   nome: neigh[2][2]   val: 0   fixo: 1   prob: 1
num: 7   nome: neigh[3][2]   val: 1   fixo: 1   prob: 1
num: 8   nome: neigh[4][2]   val: 1   fixo: 1   prob: 1
num: 9   nome: neigh[1][3]   val: 1   fixo: 1   prob: 1
num: 10   nome: neigh[2][3]   val: 1   fixo: 1   prob: 1
num: 11   nome: neigh[3][3]   val: 0   fixo: 1   prob: 1
num: 12   nome: neigh[4][3]   val: 1   fixo: 1   prob: 1
num: 13   nome: neigh[1][4]   val: 1   fixo: 1   prob: 1
num: 14   nome: neigh[2][4]   val: 1   fixo: 1   prob: 1
num: 15   nome: neigh[3][4]   val: 1   fixo: 1   prob: 1
num: 16   nome: neigh[4][4]   val: 0   fixo: 1   prob: 1
num: 17   nome: vc[1][1]   val: 0   fixo: 0   prob: 1
num: 18   nome: vc[2][1]   val: 0   fixo: 0   prob: 1
num: 19   nome: vc[3][1]   val: 0   fixo: 0   prob: 1
num: 20   nome: vc[4][1]   val: 0   fixo: 0   prob: 1
num: 21   nome: vc[1][2]   val: 0   fixo: 0   prob: 1
num: 22   nome: vc[2][2]   val: 0   fixo: 0   prob: 1
num: 23   nome: vc[3][2]   val: 0   fixo: 0   prob: 1
num: 24   nome: vc[4][2]   val: 0   fixo: 0   prob: 1
num: 25   nome: vc[1][3]   val: 0   fixo: 0   prob: 1
num: 26   nome: vc[2][3]   val: 0   fixo: 0   prob: 1
num: 27   nome: vc[3][3]   val: 0   fixo: 0   prob: 1
num: 28   nome: vc[4][3]   val: 0   fixo: 0   prob: 1
num: 29   nome: vc[1][4]   val: 0   fixo: 0   prob: 1
num: 30   nome: vc[2][4]   val: 0   fixo: 0   prob: 1
num: 31   nome: vc[3][4]   val: 0   fixo: 0   prob: 1
num: 32   nome: vc[4][4]   val: 0   fixo: 0   prob: 1
num: 33   nome: col[1]   val: 0   fixo: 0   prob: 1
num: 34   nome: col[2]   val: 0   fixo: 0   prob: 1
num: 35   nome: col[3]   val: 0   fixo: 0   prob: 1
num: 36   nome: col[4]   val: 0   fixo: 0   prob: 1
```

Figura 4.7.7: Neurônios Gerados para uma Instância do Problema de Coloração de Grafos com n=4.

fases necessárias para a construção deste compilador.

O capítulo seguinte apresenta o Problema da Inferência Lógica baseada em Resolução e como este foi modelado utilizando restrições. Esta modelagem será traduzida para a linguagem do SATyrus para sua posterior compilação e simulação.

Capítulo 5

Resultados: Compilando e Simulando a ARQ-PROP-II

Neste capítulo apresentaremos ARQ-PROP-II, um sistema de inferência conexionista que foi modelado utilizando restrições booleanas. Isto nos servirá para fazer um teste maior do compilador do SATyrus. A Seção 5.2 descreve seus conjuntos lógicos de neurônios e a Secão 5.3 os conjuntos de restrições que definem suas interconexões. Por último os resultados são mostrados na Seção 5.4.

5.1 ARQ-PROP-II

Tomar decisões, fazer reflexões, encontrar respostas aos nossos atos ou dos outros são atividades comuns em nossas vidas. Essas atividades sã realizadas a partir do conhecimento que temos sobre um assunto, o qual pode ser até incompleto e inconsistente. Esta operação mental de derivar um conhecimento novo a partir de outros designa-se como **inferência** [14]. A inferência é então uma atividade humana importante e, como tal, tem sido objeto de estudo no campo da Inteligência Artificial.

A ARQ-PROP-II é uma máquina neural proposta por Lima [20] [18] que realiza inferências por refutação baseando-se na aplicação do **Princípio de Resolução** [27] à lógica proposicional. Devido à sua arquitetura e modelagem não é necessário que as cláusulas sejam aprendidas por meio de exemplos ou pré-codificadas em forma de restrições, como foi proposto na ARQ-PROP-I [19], a primeira versão desta arquitetura. A ARQ-PROP-II realiza raciocínio monotônico com conhec-

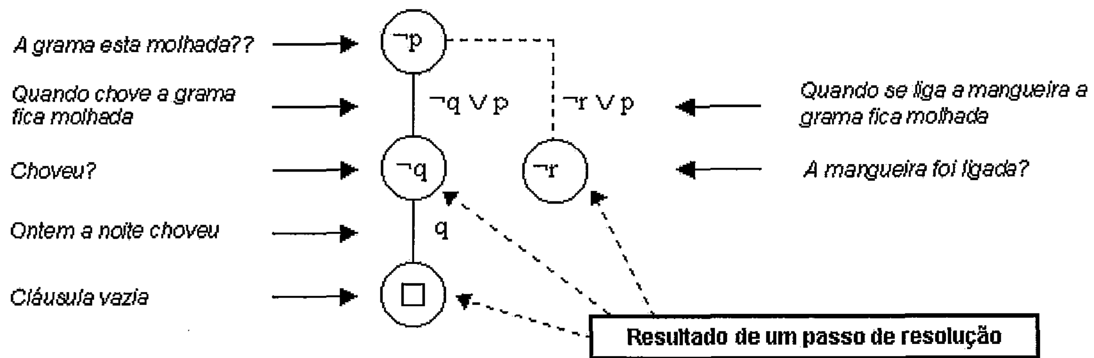


Figura 5.1.1: Passos de Processo de Resolução.

imento completo* e incompleto já que tem a capacidade de hipotetizar novas cláusulas. Os resultados experimentais obtidos neste trabalho utilizam somente conhecimento completo [28].

Para exemplificar o processo de inferência mostramos a seguir um conjunto de cláusulas (com suas respectiva representação em lógica) que representam o conhecimento que se tem sobre o assunto.

- *Quando chove a grama fica molhada.* $p \leftarrow q$
- *Quando se liga a mangueira a grama fica molhada.* $p \leftarrow r$
- *Ontem a noite choveu.* q
- *A grama está molhada.* p

A nossa pergunta é: *a grama está molhada?.*

Como sera um processo por refutação, o atomo p é negado: $\neg p$. A Figura 5.1.1 mostra cada um dos passos do processo de resolução.

5.2 Estruturas Neuronais da ARQ-PROP-II

A ARQ-PROP-II é composta por onze matrizes de neurônios que são de uma, duas ou três dimensões. Cada uma tem uma função específica e sua relação com o resto

*Entende-se como conhecimento completo quando não será necessário formular hipóteses para obter uma prova para uma determinada fórmula

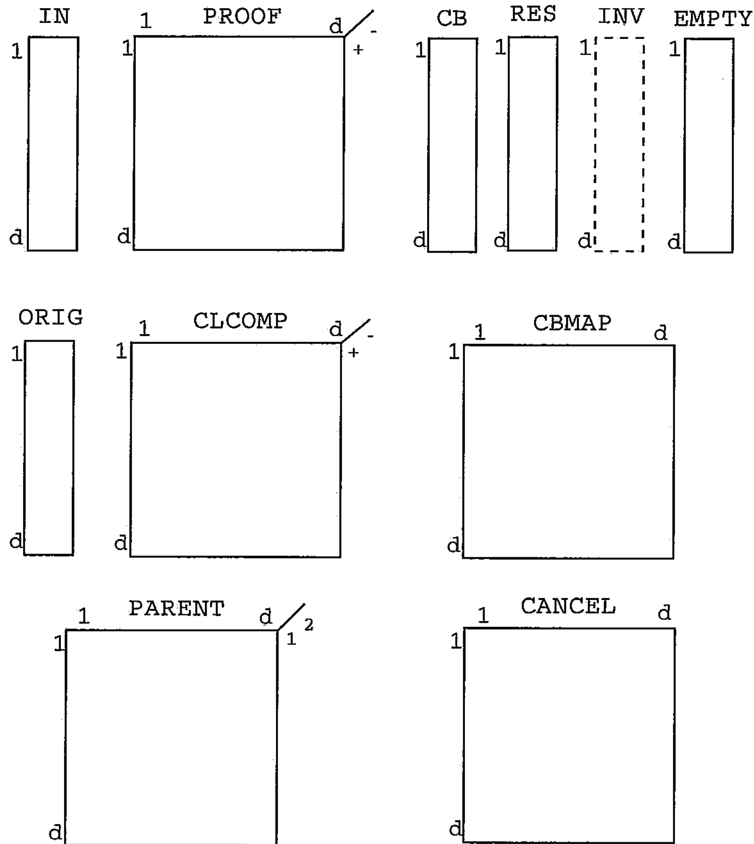


Figura 5.2.2: Conjuntos de Neurônios da ARQ-PROP-II.

de matrizes está dada pela função de energia gerada a partir das restrições. A Figura 5.2.2 mostra cada uma das matrizes que formam parte desta arquitetura. A matriz INV está em linha pontilhada porque não será utilizada para gerar resultados neste trabalho.

ARQ-PROP-II tem uma área de prova (PROOF) onde cada fila de neurônios representa uma cláusula que é copiada da base de cláusulas (CLCOMP) ou que é resultado de um passo de resolução. A linha mais inferior ou a mais superior, dependendo da configuração que se utilize, representa a cláusula vazia, na qual todos os neurônios ficam em OFF.

A seguir descreve-se detalhadamente cada uma das matrizes:

- **IN** ($d \times 1$): Indica se uma linha forma parte da prova ou não.

  Seja in_i um neurônio desta estrutura.

i é o número da linha na área de prova.

- **PROOF** ($d \times d \times 2$): Representa a área de prova.

  Seja $proof_{ijk}$ um neurônio desta estrutura.

  i é o número de uma linha da prova.

  j é o número de uma variável proposicional que faz parte da linha de prova i.

  k é o número do sinal da variável j. 1 se é positivo (+) e 2 caso contrário (-).

- **CB** ($d \times 1$): Indica se uma linha da área de prova é uma cópia da base de cláusulas ou não.

  Seja cb_i um neurônio desta estrutura.

  i é o número da linha na área de prova.

- **RES** ($d \times 1$): Indica se uma linha da área de prova é o resultado de um passo da resolução ou não.

  Seja res_i um neurônio desta estrutura.

  i é o número da linha na área de prova.

- **EMPTY** ($d \times 1$): Indica se uma linha da área de prova é ou não a cláusula vazia.

  Seja $empty_i$ um neurônio desta estrutura.

  i é o número da linha na área de prova.

- **ORIG** ($d \times 1$): Indica se uma linha do CLCOMP é original ou inventada.

  Seja $orig_i$ um neurônio desta estrutura.

  i é o número da linha no CLCOMP.

- **CLCOMP** ($d \times d \times 2$): Representa a base de cláusulas.

  Seja $clcomp_{ijk}$ um neurônio desta estrutura.

  i é o número de uma cláusula.

j é o número de uma variável proposicional que faz parte da cláusula i.

k é o número do sinal da variável j. 1 se é positivo (+) e 2 caso contrário (-).

- **CBMAP** ($d \times d$): Indica a relação entre a área de prova PROOF e a base de cláusulas CLCOMP.

  Seja $cbmap_{ij}$ um neurônio desta estrutura.

  i é o número da linha na área de prova.

  j é o número da cláusula que está na linha i.

- **PARENT** ($d \times d \times 2$): Indica os pais de uma linha resultante num passo de resolução na área de prova. Uma linha resultante tem sempre dois pais.

  Seja $parent_{ijk}$ um neurônio desta estrutura.

  i é o número de uma linha pai na área de prova.

  j é o número da linha do filho da linha i na área de prova.

  k é o número do pai (1/2).

- **CANCEL** ($d \times d$): Indica qual variável proposicional da área de prova foi cancelada.

  Seja $cancel_{ij}$ um neurônio desta estrutura.

  i é o número da linha na área de prova.

  j é o número da variável proposicional cancelada na linha i.

5.3 Conjunto de Restrições da ARQ-PROP-II

Na modelagem inicial da ARQ-PROP-II dois níveis de penalidades foram propostos. Durante o processo de experimentação foram testadas cada uma das restrições separadamente e todas trabalharam corretamente. O problema surgia ao testá-las em conjunto: algumas inibiam o funcionamento das outras, o qual obviamente não permitia o correto funcionamento do modelo.

Depois de vários testes concluiu-se que a atribuição dos níveis de penalidades não era a melhor e formulou-se então uma outra forma de agrupar as restrições. Na nova formulação as restrições de integridade dividem-se em 3 grupos:

Nível	Descrição
0	Para as restrições de otimalidade.
1	Para as restrições com existencial (ou com ou exclusivo).
2	Para as restrições sem existencial (ou sem ou exclusivo).
3	Para as restrições WTA.

Tabela 5.1: Níveis de Penalidade.

- Restrições com existencial: nas quais pelo menos uma das cláusulas do conseqüente é verdadeira.
- Restrições WTA (*Winner-Takes-All*): nas quais só um dos átomos está ON;
- Restrições sem existencial: o restante das restrições.

Definiu-se então quatro níveis de prioridade para problemas com restrições de otimalidade e três para aqueles, que como este, só tem restrições de integridade. A Tabela 5.1 mostra estes níveis.

A seguir o conjunto de restrições da ARQ-PROP-II:

1. Restrições das estruturas IN e PROOF

    - Se uma linha está na prova, então é uma cópia da base de cláusulas ou o resultado de um passo de resolução.

      $\forall_i 1 \leq i \leq d^{\dagger} : IN_i \rightarrow CB_i \vee RES_i$

      Nível de penalidade=2

    - Se uma linha está na prova e esta não é a cláusula vazia, então esta pode ser pai de outra na área de prova. Somente as linhas que não contêm a cláusula vazia podem ser pais de outras.

      $\forall_{i,j,k} 1 \leq i \leq d, 1 \leq j \leq d, k \in \{1,2\} : IN_i \wedge \neg EMPTY_i \rightarrow PARENT_{ijk}$

      Nível de penalidade=1

    - A linha que contém a cláusula vazia não pode ser pai de outra na área de prova.

† d pertence aos números naturais

$\forall_{i,j,k} 1 \leq i \leq d, 1 \leq j \leq d, k \in \{1,2\} : IN_i \wedge EMPTY_i \rightarrow \neg PARENT_{ijk}$

Nível de penalidade=2

2. Restrições de uma cláusula

   - Uma linha i da área de prova que é cópia da cláusula j da base de cláusulas deve estar associada a esta mediante CBMAP.

     $\forall_{i,j} 1 \leq i \leq d, 1 \leq j \leq d : CB_i \rightarrow CBMAP_{ij}$

     Nível de penalidade=1

   - Uma instância da cláusula j de CB deve ser copiada da base de cláusulas original.

     $\forall_{i,j} 1 \leq i \leq d, 1 \leq j \leq d : ORIG_j \wedge CBMAP_{ij} \rightarrow CB_i$

     Nível de penalidade=2

3. Restrições da sintaxe de uma cláusula

   Somente os literais que pertencem à cláusula i na base de cláusulas devem estar presentes na linha de prova que representa a cláusula i.

   - $\forall_{i,j,k,s} 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, s \in \{+,-\} : CBMAP_{ij} \wedge CLCOMP_{jks} \rightarrow PROOF_{iks}$

     Nível de penalidade=2

   - $\forall_{i,j,k,s} 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, s \in \{+,-\} : CBMAP_{ij} \wedge \neg CLCOMP_{jks} \rightarrow \neg PROOF_{iks}$

     Nível de penalidade=2

4. Restrições dos passos da resolução

   - Cada linha resultante de um passo de resolução deve ter dois pais.

     $\forall_{i,j,k,l,m} 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, l,m \in \{+,-\}; j \neq k, m \neq l : RES_i \rightarrow (PARENT_{jil} \wedge PARENT_{kim})$

     Nível de penalidade=2

   - Deve haver um par cancelado de literais por passo de resolução.

     $\forall_{i,k} 1 \leq i \leq d, 1 \leq k \leq d : RES_i \rightarrow CANCEL_{ik}$

     Nível de penalidade=1

5. Restrições da estrutura PARENT

- As linhas j e k envolvidas em um passo de resolução devem conter um membro do par cancelado.

  $\forall_{i,j,k,l,m,n} 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, 1 \leq l \leq d, m, n \in \{+,-\}, k \neq j, l \neq m : PARENT_{ji1} \wedge PARENT_{ki2} \wedge PROOF_{jlm} \wedge PROOF_{kln} \rightarrow CANCEL_{il}$

  Nível de penalidade=2

- Somente as linhas que são resultado de um passo de resolução têm pais.

  $\forall_{i,j,k} 1 \leq i \leq d, 1 \leq j \leq d, k \in \{1,2\} : \neg RES_i \rightarrow \neg PARENT_{jik}$

  Nível de penalidade=2

- A linha resultante de um passo de resolução deve formar parte da prova.

  $\forall_{i,j,k} 1 \leq i \leq d, 1 \leq j \leq d, k \in \{1,2\} : PARENT_{j,i,k} \rightarrow \neg IN_i$

  Nível de penalidade=2

6. Restrições da composição do resolvente

- Copia ao resolvente todos os literais dos pais que não foram cancelados.

  $\forall_{i,j,k,l,s} 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, l \in \{1,2\}, s \in \{+,-\} : PARENT_{jil} \wedge PROOF_{jks} \wedge \neg CANCEL_{ik} \rightarrow PROOF_{iks}$

  Nível de penalidade=2

- Copia ao resolvente somente os literais que estavam nos pais.

  $\forall_{i,j,k,l,m,n,p} 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, 1 \leq l \leq d, m, n \in \{1,2\}, p \in \{+,-\}, j \neq k, m \neq n : PARENT_{jim} \wedge PARENT_{kin} \wedge \neg PROOF_{jlp} \wedge \neg PROOF_{klp} \rightarrow \neg PROOF_{ilp}$

  Nível de penalidade=2

- Não copia o par cancelado.

  $\forall_{i,j,s} 1 \leq i \leq d, 1 \leq j \leq d, s \in \{+,-\} : CANCEL_{ij} \rightarrow \neg PROOF_{ijs}$

  Nível de penalidade=2

7. Restrições da cláusula vazia

- Uma linha de prova é vazia se não tem literais.

  $\forall_{i,j,s} 1 \leq i \leq d, 1 \leq j \leq d, s \in \{+,-\} : EMPTY_i \rightarrow \neg PROOF_{ijs}$

  Nível de penalidade=2

- Uma linha da prova tem literais se não é vazia.

  $\forall_{i,j,s} 1 \leq i \leq d, 1 \leq j \leq d, s \in \{+,-\} : PROOF_{ijs} \rightarrow \neg EMPTY_i$

  Nível de penalidade=2

- A cláusula vazia deve ser sempre resultado de um passo de resolução.

  $\forall_i 1 \leq i \leq d : EMPTY_i \rightarrow \neg RES_i$

8. Restrições WTA

- Um símbolo proposicional deve aparecer só uma vez por linha na área de prova (PROOF).

  $\forall_{i,j,s,k} 1 \leq i \leq d, 1 \leq j \leq d, s \in \{+,-\}, k \in \{+,-\}; s \neq k : (PROOF_{i,j,s} \rightarrow \neg PROOF_{i,j,k})$

- Uma linha na área de prova tem só uma justificativa (ou é cópia da base de cláusulas ou resultado de um passo de resolução).

  $\forall_i; 1 \leq i \leq d : (CB_i \rightarrow \neg RES_i)$

- Pode ser copiada só uma cláusula da base de cláusulas por linha na área de prova.

  $\forall_{i,j,k} 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d; j \neq k : (CBMAP_{i,j} \rightarrow \neg CBMAP_{i,k})$

- Uma linha da área de prova tem só um pai 1 e só um pai 2.

  $\forall_{i,j,k,l} 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, l \in \{1,2\}; j \neq k : (PARENT_{j,i,l} \rightarrow \neg PARENT_{k,i,l})$

- Uma linha da área de prova pode ser um dos pais de outra só uma vez.

  $\forall_{i,j,n,k,l} 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq n \leq d, k \in \{1,2\}, l \in \{1,2\}; i \neq n : (PARENT_{j,i,k} \rightarrow \neg PARENT_{j,n,l})$

- As linhas pais envolvidas num passo de resolução devem ser diferentes.

  $\forall_{i,j,k,l} 1 \leq i \leq d, 1 \leq j \leq d, k \in \{1,2\}, l \in \{1,2\}; k \neq l : (PARENT_{j,i,k} \rightarrow \neg PARENT_{j,i,l})$

- Por passo de resolução, só um par de literais pode ser cancelado.
  $\forall_{i,j,k} 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d; j \neq k : (CANCEL_{i,j} \rightarrow \neg CANCEL_{i,k})$

Modificaram-se as duas primeiras restrições do grupo 6 para evitar que fosse necessária a utilização de um procedimento de eliminação de proposições redundantes. Tal foi obtido através da introdução do requerimento de uma desigualdade entre alguns pares de índices. Além disso, no grupo 5 uma restrição foi adicionada para reforçar o correto funcionamento da matriz IN. A seguir descrevemos esta última restrição junto com as modificadas:

- Restrições modificadas (em negrito as modificações feitas):
  - $\forall_{i,j,k,l,s} 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, l \in \{1,2\}, s \in \{+,-\}, \mathbf{j} \neq \mathbf{i} : PARENT_{jil} \wedge PROOF_{jks} \wedge \neg CANCEL_{ik} \rightarrow PROOF_{iks}$
  - $\forall_{i,j,k,l,m,n,p} 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, 1 \leq l \leq d, m, n \in \{1,2\}, p \in \{+,-\}, j \neq k, m \neq n, \mathbf{k} \neq \mathbf{i} : PARENT_{jim} \wedge PARENT_{kin} \wedge \neg PROOF_{jlp} \wedge \neg PROOF_{klp} \rightarrow \neg PROOF_{ilp}$

- Restrição nova:
  - O pai de uma linha da prova deve também formar parte da prova.
    $\forall_{i,j,k} 1 \leq i \leq d, 1 \leq j \leq d, k \in \{1,2\} : PARENT_{j,i,k} \rightarrow IN_j$

5.4 Resultados

Todos os testes foram feitos utilizando-se um código compilado em C++ ver. 3.4.5 em plataforma Linux. No Apêndice D encontra-se o código fonte da ARQ-PROP-II que ingressa ao compilador do SATyrus. Os arquivos para preencher as matrizes de neurônios variam dependendo do grupo de cláusulas que está sendo testado. Finalmente, os valores da temperatura inicial (T_i) e final (T_f), e do fator de atualização da temperatura (F) para o processo de simulação foram:

$T_i = 10000$

$T_f = 1$

$F = 0,99$

5.4.1 Teste 1

Para este teste utilizaremos o conjunto de cláusulas do nosso exemplo da grama molhada:

$$p \leftarrow q$$
$$p \leftarrow r$$
$$q$$
Meta: p

O número de filas utilizadas na área de prova foram 6. A seguir as saídas do compilador:

Número total de neurônios da rede: 318.
Número total de parcelas da função de energia: 15744.
Valor do nível de penalidade 0: 1.
Valor do nível de penalidade 1: 8396.
Valor do nível de penalidade 2: 69015140.
Valor final da energia: 169.

Neste exemplo só um caminho leva à cláusula vazia. A Figura 5.4.3 mostra este caminho e em linha pontilhada aquele que por falta de conhecimento não foi finalizado. Atingir o ótimo global não foi uma tarefa fácil para o simulador, pelo que vários pontos de início foram testados através da opção de início aleatório do compilador.

A Figura 5.4.4 mostra o comportamento da Função de Energia e na Figura 5.4.5 mostra-se a configuração final das matrizes da ARQ-PROP-II. Os neurônios de cor preta são a entrada ao compilador e os de cor cinza são o resultado final. Pode-se notar que existem neurônios em ON na área de prova (PROOF) que não representam nenhuma das cláusulas de entrada. Estes neurônios, especificamente os da linha 2, não devem ser considerados devido a que na matriz IN esta linha ficou em OFF. O fato destes neurônios ficarem em ON não influencia o valor final da função de energia.

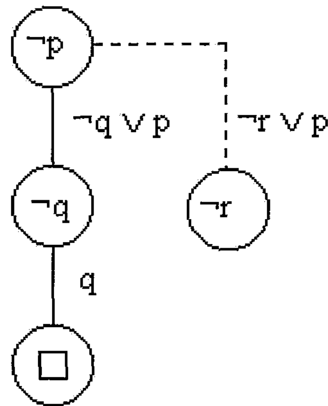


Figura 5.4.3: Caminho que leva à Cláusula Vazia (Teste 1).

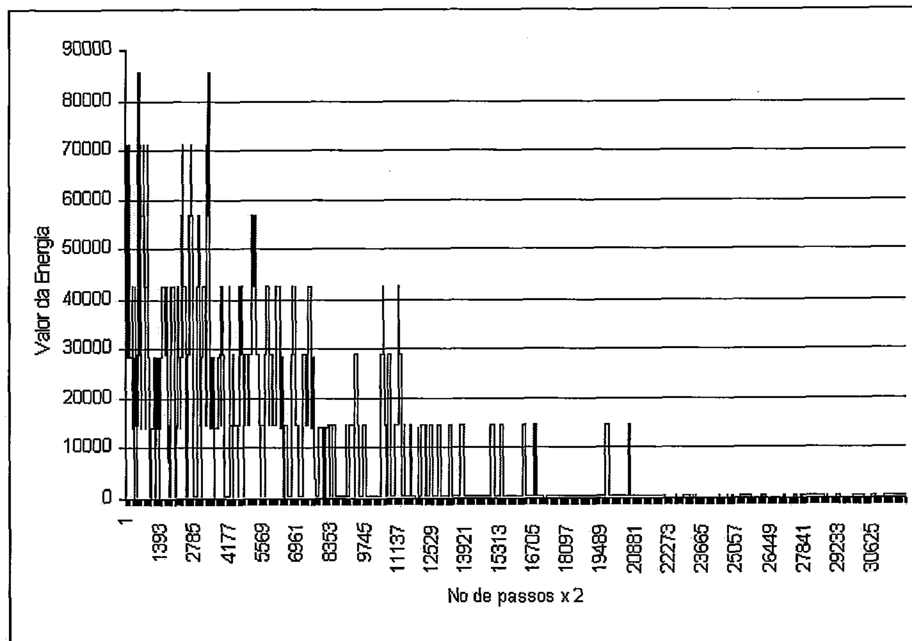


Figura 5.4.4: Trajetória da Função de Energia (Teste 1).

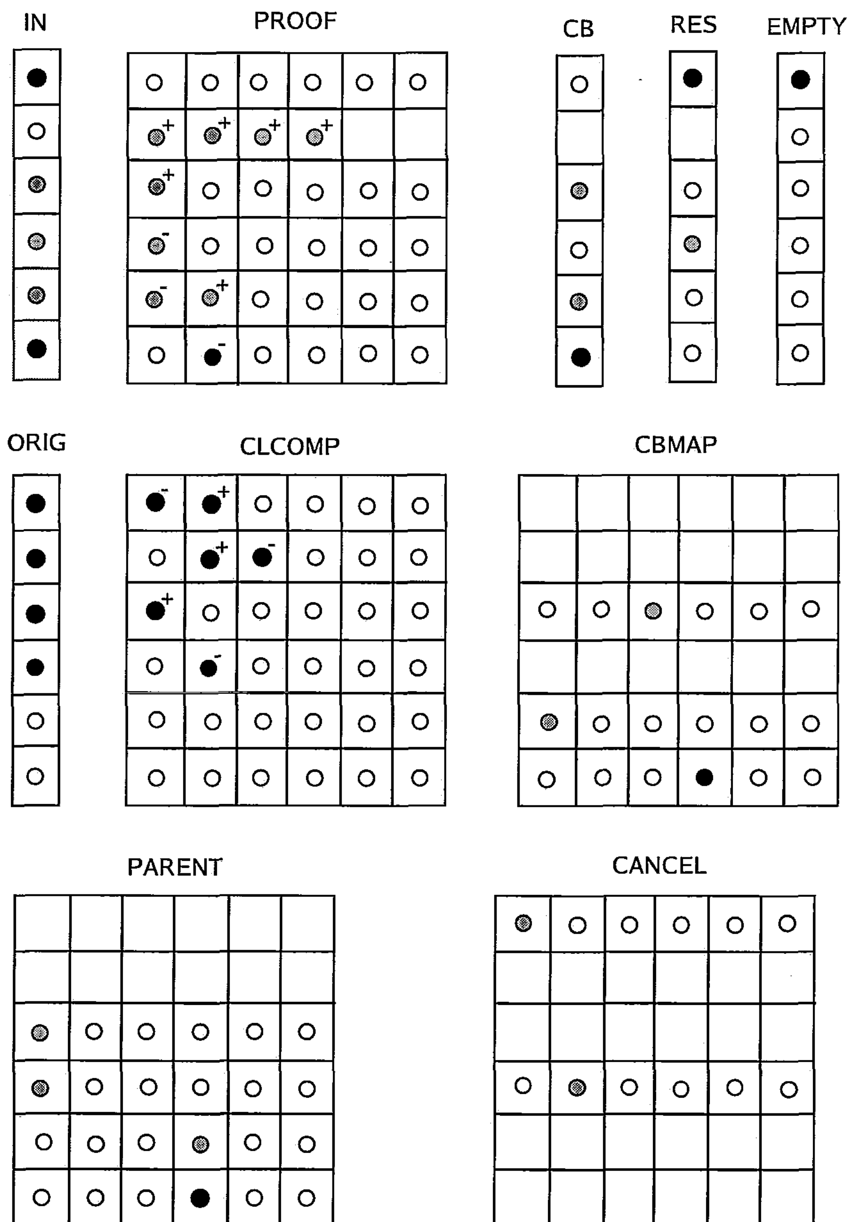


Figura 5.4.5: Configuração Final da ARQ-PROP-II (Teste 1).

5.4.2 Teste 2

Para este teste incrementamos uma cláusula na base de conhecimento do Teste 1:

- *A mangueira ficou ligada:* r

$$\begin{aligned} & p \leftarrow q \\ & p \leftarrow r \\ & q \\ & r \\ \text{Meta:}\quad & p \end{aligned}$$

O número de filas utilizadas na área de prova foram 6. A seguir as saídas do compilador:

Número total de neurônios da rede: 318.
Número total de parcelas da função de energia: 15744.
Valor do nível de penalidade 0: 1.
Valor do nível de penalidade 1: 8396.
Valor do nível de penalidade 2: 69015140.
Valor final da energia: 169.

Para este exemplo o conhecimento faltante no Teste 1 foi completado. Devido a isto, dois caminhos levam à cláusula vazia, tal como se pode apreciar na Figura 5.4.6. Devido à dificuldade para atingir o ótimo global, uma das respostas foi encontrada diretamente usando o simulador e a outra (em linha pontilhada) foi obtida pela avaliação da função de energia nesse ponto.

A Figura 5.4.7 mostra o comportamento da Função de Energia. Na Figura 5.4.8 mostra-se a primeira configuração final das matrizes do ARQ-PROP II e na Figura 5.4.9 a segunda possível configuração. Pode-se observar, como no teste anterior, que existem neurônios em ON na área de prova que como já tinha-se dito não afetam o valor final da função de energia. Mas, diferentemente do teste anterior existem também vários neurônios em ON na matriz CANCEL. Estes neurônios tampouco influenciam a função de energia, devido a que só se consideram as linhas da matriz RES que tenham neurônios em ON.

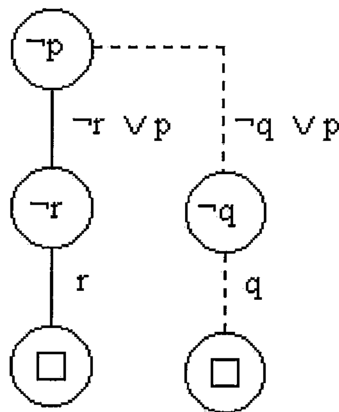


Figura 5.4.6: Caminhos que levam à Cláusula Vazia (Teste2).

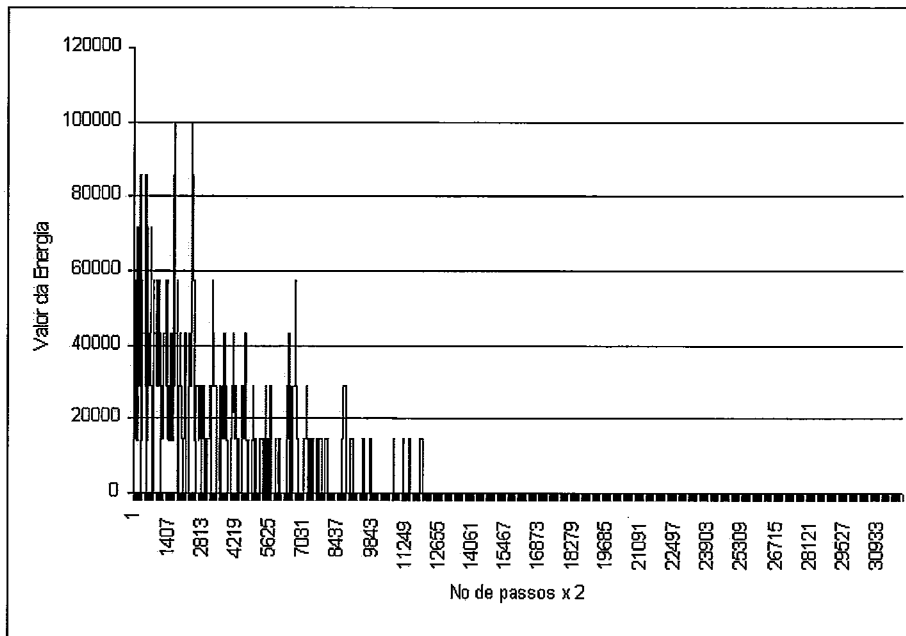


Figura 5.4.7: Trajetória da Função de Energia (Teste 2).

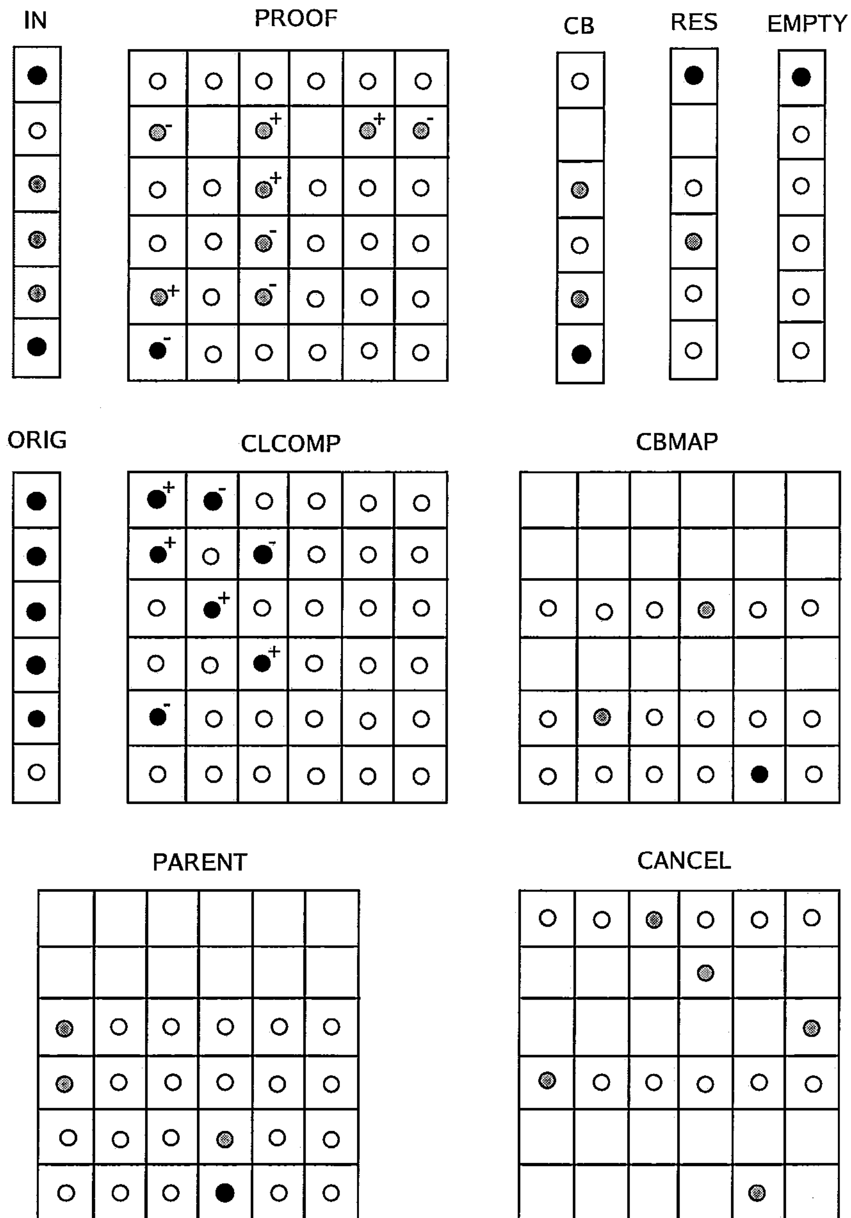


Figura 5.4.8: Primeira Configuração Final do ARQ-PROP-II (Teste 2).

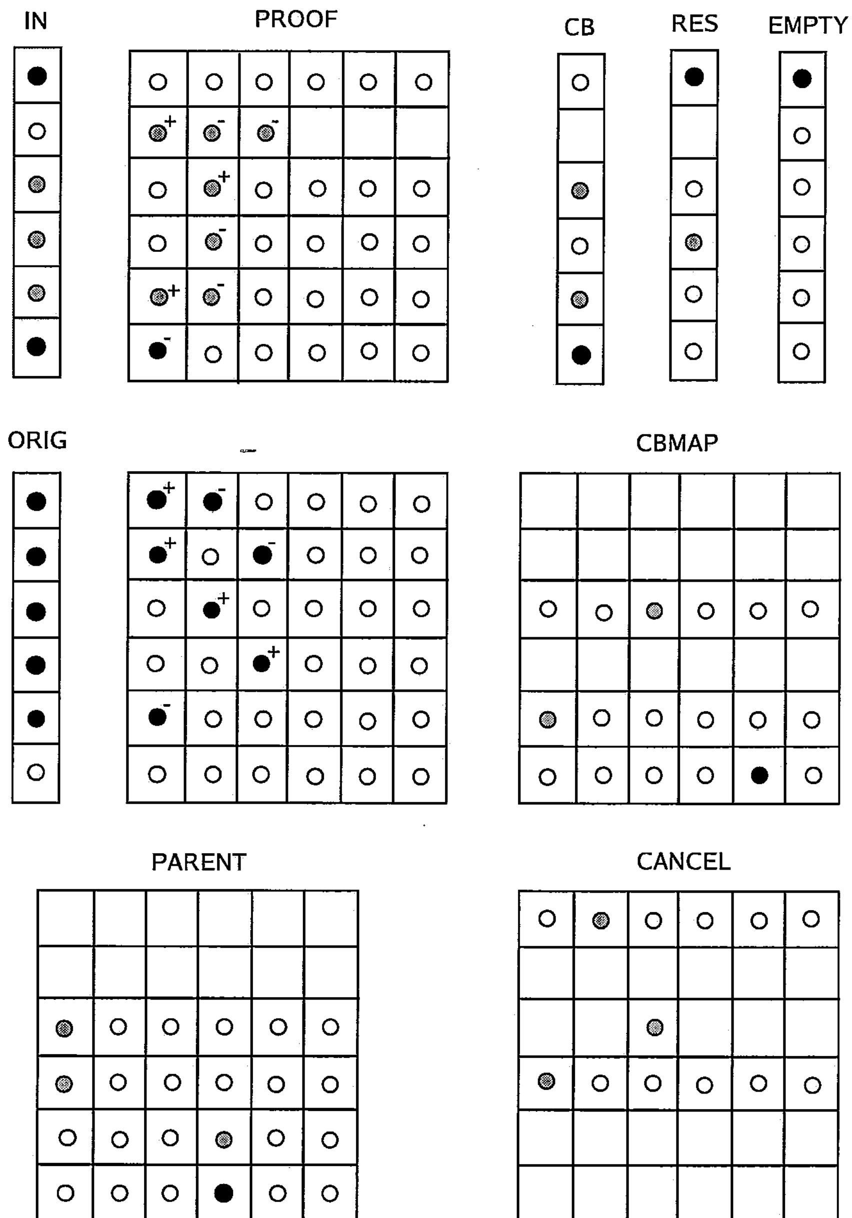


Figura 5.4.9: Segunda Configuração Final do ARQ-PROP-II (Teste 2).

5.4.3 Teste 3

Para este teste modificamos duas cláusulas e as combinamos em uma só:

- *Se chove e se liga a mangueira, a grama fica molhada:* $p \leftarrow q \wedge r$

$p \leftarrow q \wedge r$

q

r

Meta: p

O número de filas utilizadas na área de prova foram 6. A seguir as saídas do compilador:

Número total de neurônios da rede: 552.

Número total de parcelas da função de energia: 49216.

Valor do nível de penalidade 0: 1.

Valor do nível de penalidade 1: 45748.

Valor do nível de penalidade 2: 240817492.

Valor final da energia: 433.

Neste exemplo existem dois caminhos possíveis para alcançar a cláusula vazia. A dificuldade para atingir o ótimo foi maior ainda, pelo que o tempo para encontrar uma configuração de início que ajudasse a alcançá-lo foi ainda maior. A Figura 5.4.10 ilustra os dois caminhos e em linha pontilhada aquele que foi diretamente avaliado na função de energia.

Na Figura 5.4.12 mostra-se a primeira configuração final das matrizes do ARQ-PROP II e na Figura 5.4.13 a segunda possível configuração. Finalmente, na Figura 5.4.11 se mostra a trajetória da função de energia.

5.5 Comentários

Este capítulo introduziu a ARQ-PROP-II para testar o desempenho do compilador do SATyrus em exemplos com complexidade razoavelmente grande. Os experimentos demonstraram que a geração da energia é correta, mas que a dificuldade para

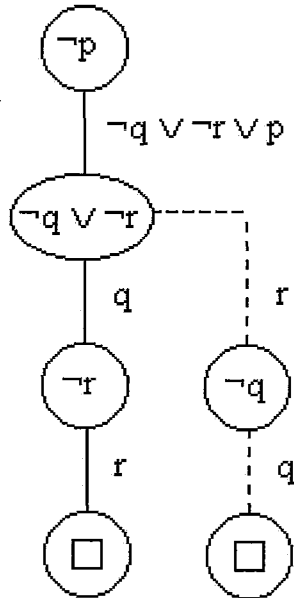


Figura 5.4.10: Caminhos que levam à Cláusula Vazia (Teste3).

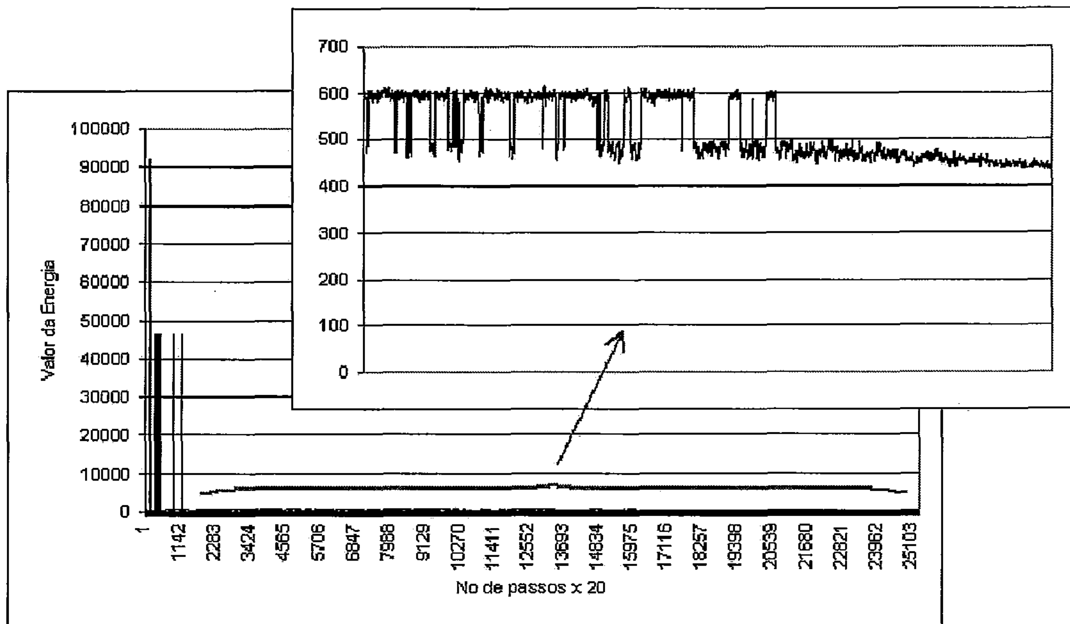


Figura 5.4.11: Trajetória da Função de Energia (Teste 3).

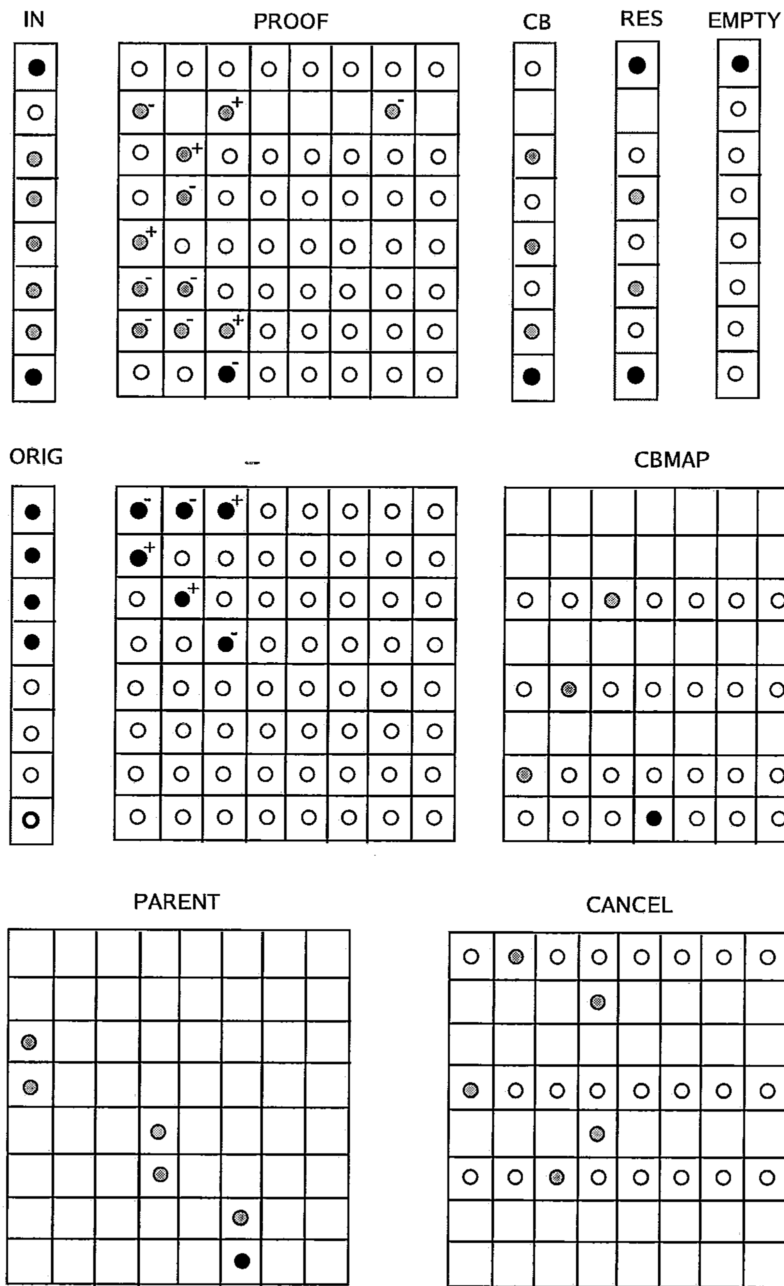


Figura 5.4.12: Primeira Configuração Final do ARQ-PROP-II (Teste 3).

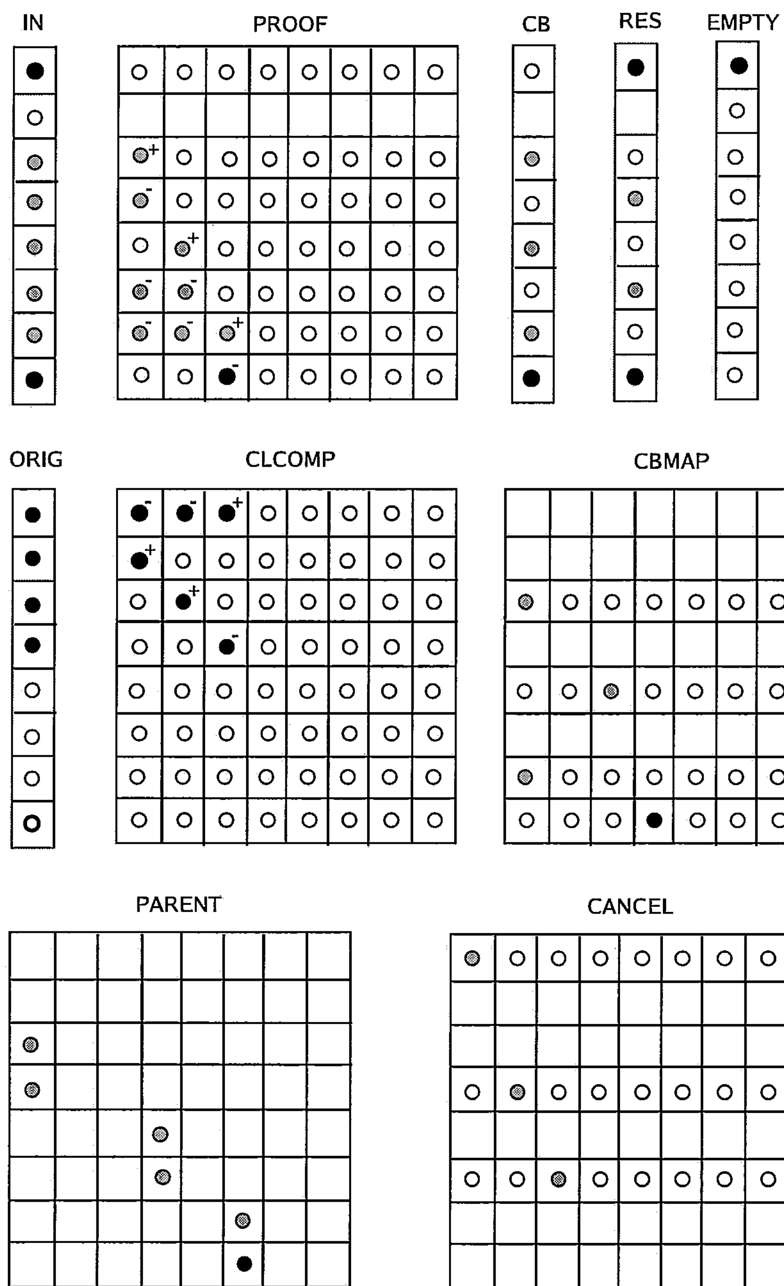


Figura 5.4.13: Segunda Configuração Final do ARQ-PROP-II (Teste 3).

atingir o ótimo global vai crescendo com as dimensões da rede gerada. A solução foi começar a simulação desde diferentes pontos de início aleatórios.

O Apêndice E contém testes para algumas das possíveis entradas ao compilador. Em todos os testes as saídas do compilador foram corretas, isto valida seu funcionamento e garante saídas sem erro.

Capítulo 6

Conclusões

Após ter realizado os experimentos e descrito os resultados obtidos, este capítulo resume as conclusões obtidas ao longo do desenvolvimento do trabalho, e apresenta uma breve discussão com base em alguns trabalhos correlatos. Finalmente, trabalhos futuros a serem desenvolvidos com base nesta pesquisa são apresentados.

6.1 Resumo

Este trabalho apresentou:

- Uma linguagem para a especificação de problemas que possam ser descritos através de restrições escritas como cláusulas lógicas.
- O compilador do SATyrus, que transforma um problema escrito nesta linguagem em uma função de energia equivalente.
- Um conjunto de testes, onde se combinam as possíveis entradas ao compilador, para validar seu correto funcionamento.
- A compilação e simulação do TSP e do problema de Coloração de Grafos, como exemplos básicos do funcionamento do SATyrus.
- A compilação e simulação da ARQ-PROP-II, como um exemplo de maior tamanho e complexidade.

A partir dos experimentos e resultados conclui-se que:

- Um problema que possa ser descrito usando restrições lógicas, torna-se um candidato idôneo para ser resolvido pelo SATyrus.

- Quanto mais cresce em complexidade um problema e pelo tanto em quantidade de neurônios, atingir o mínimo global se torna mais difícil e mais custoso em tempo.

- As possibilidades de atingir o ótimo global aunmentam mais quando se começa a simulação desde diferentes pontos do espaço de busca.

- A saída gerada pelo compilador foi correta em todos os casos testados, já que o mínimo ou mínimos globais das funções de energia corresponderam às soluções ótimas dos problemas testados.

- O compilador facilita o processo de conversão das cláusulas para sua função de energia equivalente e garante que não existam erros nos cálculos e procedimentos, especialmente quando o número de restrições cresce e a complexidade de cada uma também. O resultado é dado com rapidez e exatidão.

6.2 Discussão

Na Sub-seção 1.1 foram apresentados alguns trabalhos correlatos. Nesta seção, uma comparação destes com o SATyrus será feita.

A notação Z é indiscutivelmente uma linguagem muito poderosa para especificar sistemas, não obstante a transformação destas especificações em um programa que resolva o problema se torna complicado. O SATyrus possui também uma linguagem de especificação e além disso um compilador que traduzirá o conjunto de especificações numa Função de Energia. Esta pode ser representada por uma rede neural Hopfield de Alta Ordem, a qual, depois de um processo de busca, produz como saída a solução esperada.

Diferentemente do NEUCOMP, o SATyrus permite especificar diferentes conjuntos de restrições que representam diferentes problemas e, conseqüentemente, redes neurais de diferentes arquiteturas e tamanhos são criadas. Isto significa que, a pesar de não gerar diferentes modelos de redes neurais, o SATyrus está um passo adiante no processo resolução de problemas. Já em comparação com o NEUCOMP2, o SATyrus ainda não permite criar redes neurais paralelas. Mas, o processo de gridificação do SATyrus é um projeto a médio prazo.

Assim como AMPL e MathSAT, o SATyrus foi desenvolvido para resolver problemas de otimização, e além disso pode também resolver qualquer um que possa ser especificado utilizando restrições. Em comparação com o MathSAT3 o compilador do SATyrus tem menos flexibilidade na entrada, já que estas restrições tem que ser escritas em Forma Normal Conjuntiva. Por outro lado, a saída do compilador do SATyrus é suficientemente independente, o que permite poder utilizar outros aplicativos de otimização, como o Matlab [32] por exemplo, para o processo de simulação. Adicionalmente, ao converter a formulação de um problema para minimização de energia, a plataforma SATyrus fornece uma formulação exata para o problema em questão.

6.3 Trabalhos Futuros

A seguir descrevemos alguns trabalhos futuros baseados nesta pesquisa:

- A implementação de uma interface gráfica com os recursos apropriados para os usuários do SATyrus.
- A implementação de um pre-processador que transforme qualquer fórmula na Forma Normal Conjuntiva, o que facilitará ao usuário a especificação dos problemas.
- A simulação do ARQ-PROP-II com conhecimento incompleto e a simulação da versão para lógica de primeira ordem.
- A melhora da eficiência do compilador em termos de complexidade de armazenamento e velocidade de criação das parcelas.
- Incrementar a funcionalidade do compilador, implementando um módulo que gere redes neurais paralelas, isto devido a que o tempo consumido durante a simulação cresce muito com problemas complexos e o ideal é criar uma rede que rode paralelamente.

Referências Bibliográficas

[1] Aho A. and Sethi R. and Ullman J. *Compilers: Principles, Techniques and Tools.* 1986.

[2] Bozzano M. and Bruttomesso R. and Cimatti A. and Junttila T. and Van Rossum P. and Schulz S. and Sebastiani R. MATHSAT: Tight Integration of SAT and Mathematical Decision Procedures. *Journal of Automated Reasoning,* 2005.

[3] Bozzano M. and Bruttomesso R. and Cimatti A. and Junttila T. and Van Rossum P. and Schulz S. and Sebastiani R. The MathSAT 3 System. 3632:315–321, 2005.

[4] Cook, S. The Complexity of Theorem Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing,* pages 151–158, 1971.

[5] Cormen T. and Leiserson C. and Rivest R. and Stein C. *Introduction to Algorithms, Second Edition.* The MIT Press Cambridge, Massachusetts London, England, 2001.

[6] Evans D. J. and Sulaiman M. N. . NEUCOMP: Neural Network Compiler. *International Journal of Computer Mathematics,* 54, 1994.

[7] Evans D. J. and Sulaiman M. N. . NEUCOMP2 - Parallel Neural Network Compiler. *Malaysian Journal of Computer Science,* 9(2):54–70, December 1996.

[8] Evans D. J. and Sulaiman M. N. Solving Optimisation Problems using NEUCOMP: A Neural Network Compiler. *International Journal of Computer Mathematics,* 62:1–21, 1996.

[9] Fourer R. and Gay D. and Kernighan B. A Modeling Language for Mathematical Programming. 54:519–554, 1990.

[10] Free Software Foundation. Bison, 2005. `http://www.gnu.org/software/bison/`.

[11] Free Software Foundation. Flex: The Fast Lexical Analyzer, 2005. `http://flex.sourceforge.net/`.

[12] Geman, S. and Geman, D. Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 721–741, 1984.

[13] Hinton, G. and Sejnowski, T. and Ackley, D. Boltzmann Machines: Constraint Satisfaction Networks that Learn. In *Technical Report CMU-CS84 -119*, Carnegie-Mellon University, 1984.

[14] Hogger C.J. *Introduction to Logic Programming.* Academic Press Inc., 1984.

[15] Hopfield, J.J. Neural Networks and Physical Systems with Emergent Collective Computational Abilities. In *Proceedings of the National Academy of Sciences*, pages 2554–2558, USA, 1982.

[16] Kanter I. Potts-glass Models of Neural Networks. *Physical Review*, 37(7):2739–2742, April 1988.

[17] Kirkpatrick, S. and Gellat Jr. and C.D.,Vecchi and M.P. Optimization via Simulated Annealing. Number 220, pages 671–680, 1983.

[18] Lima, P. A Neural Propositional Reasoner that is Goal-Driven and Works without Pre-Compiled Knowledge. In *Sixth Brazilian Symposium on Neural Networks*, pages 361–366, 2000.

[19] Lima, P. *Logical Inference in Symmetric Neural Networks.* Tese de Doutorado, Department of Computing. Imperial College of Science, Technology and Medicine., London, UK, 2000.

[20] Lima, P. A Goal-Driven Neural Propositional Interpreter. *International Journal of Neural Systems*, 11:311–322, 2001.

[21] Lima, P. and Morveli-Espinoza M. and Pereira G. and França F. SATyrus: A SAT-based Neuro-Symbolic Arquitecture for Constraint Processing. In *Proceedings of the Fifth International Conference on Hybrid Intelligent Systems*, pages 137–142, Rio de Janeiro, Brasil, 2005.

[22] Lima, P. and Pereira G. and Morveli-Espinoza M. and França F. Mapping and Combining Combinatorial Problems into Energy Landscapes via Pseudo-boolean Constraints. In *Lecture Notes on Computer Science. Proc. BVAI05*, pages 308–317, 2005.

[23] Naur P. Revised Report on the Algorithmic Language ALGOL 60. 3:299–314, 1960.

[24] Pereira G. Mapeamento e Combinação de Problemas NP-Difíceis, através de Restrições Pseudo-booleanas para Redes Neuronais Artificiais. Dissertação de Mestrado, PESC-COPPE UFRJ, Rio de Janeiro, Brasil, Setembro 2006.

[25] Pinkas, G. The Equivalence of Energy Minimization and Propositional Calculus Satisfiability. In *Technical Report WUCS-90-03*, Washington University, 1990.

[26] Pinkas, G. *Resolution-Based Inference on Artificial Neural Networks.* Tese de Doutorado, Sever Institute of Technology, Washington University, Saint Louis, USA, 1992.

[27] Robinson J.A. A Machine-oriented Logic based on the Resolution Principle. *Journal of the ACM 12*, pages 23–41, 1965.

[28] Russel S. and Norvig P. *Artificial Intelligence A Modern Approach.* Prentice Hall, Upper Sanddle River, New Jersey 07458, 1995.

[29] Senyanrd A. and Kazmierczak E. and Sterling L. Software Engineering Methods for Neural Networks. In *Proceedings of the Tenth Asia Pacific Software Engineering Conference*, 2003.

[30] Spivey M. *The Z Notation: A Reference Manual.* Prentice Hall, International Series in Computer Science, 1992.

[31] Szwarcfiter J. *Grafos e Algoritmos Computacionais. Editora Campus.* 1984.

[32] The MathWorks. MATLAB - The Language of Technical Computing, 1994-2006. `http://www.mathworks.com/products/matlab/`.

[33] Winikoff M. and Dart P. and Kazmierczak E. Rapid Prototyping using Formal Specifications. In *Proceedings of the 21st Australasian Computer Science Conference*, pages 279–294, Perth, Australia, 1998.

Apêndice A

Manual do Compilador

Este manual detalha as características principais dos arquivos de entrada ao compilador do SATyrus, uma arquitetura neuro-simbólica orientada à resolução de problemas de otimização através do mapeamento da especificação do problema em conjuntos de restrições pseudo-booleanas.

O compilador tem como objetivo gerar uma função de energia a partir de um conjunto de restrições descritas por clausulas em Forma Normal Conjuntiva e dos atributos destas que ajudam a uma melhor definição do problema que se deseja resolver.

A.1 Estrutura do Arquivo Principal

A entrada principal do compilador é um arquivo que descreve as restrições que definem o problema. Este se compõe basicamente de 3 partes:

(i) A definição das estruturas que irão ser usadas.

(ii) Um conjunto de restrições do problema, as quais podem ser de integridade ou otimalidade.

(iii) A definição do nível das penalidades utilizadas.

A seguir detalharemos cada uma delas para uma maior compreensão.

A.1.1 Definição das Estruturas

Existem 3 tipos de estruturas:

1. Atribuição direta de um valor inteiro a uma variável: também se pode atribuir a uma variável a soma ou subtração de outras variáveis entre si ou com números inteiros. Vejamos alguns exemplos:

```
num=5;
num1=num+4;
num2=num-num1;
```

2. Estruturas que representam matrizes de neurônios de valores binários (0 ou 1). O formato da definição destas estruturas é o seguinte:

   nome_ da_ estrutura(dimensões da estrutura)

   Onde:

   - O nome da estrutura é uma cadeia alfa-numérica que começa sempre com uma letra.
   - Seja *pos* o nome de uma matriz de neurônios de duas dimensões (4×5). Existem 3 formas de se declarar as dimensões desta estrutura:

     a) Usando números: pos(4,5);

     b) Usando variáveis previamente declaradas e inicializadas. Por exemplo:

     ```
     num=4;
     num1=5;
     pos(num,num1);
     ```

     c) Usando intervalos: pos($1 \leq i \leq 4$, $1 \leq j \leq 5$);
   - A quantidade de dimensões não é restrita.

3. Estruturas que não representam matrizes de neurônios: estas matrizes têm valores inteiros que servem de coeficientes aos neurônios, neste caso além de definir a estrutura é preciso definir um arquivo texto (.txt) de onde irão ser lidos os valores. Por exemplo:

```
dist (num1,num1);
dist read from tsp1.txt;
```

```
ttsp.txt - Notepad
File Edit Format View Help
num=5;
pos(num,num);
dist (num,num);
dist read from tsp1.txt;
```

Figura 1.1.1: Definição das Estruturas Usadas em TSP

Observações

- As estruturas estão sempre separadas por ponto e vírgula (;).
- Os nomes das estruturas são cadeias alfa-numéricas que começam com uma letra.
- O nome do arquivo também é uma cadeia alfa-numérica que começa com uma letra e tem extensão *txt*.
- As dimensões das matrizes são separadas por virgulas (,).

Na Figura 1.1.1 se mostra um exemplo da definição de estruturas na linguagem do compilador do SATyrus.

A.1.2 Definição das Restrições

Grupos, penalidades e índices

- Existem dois grupos de restrições: de integridade (*integrity*) e otimalidade (*optimality*); cada restrição pertence a um deles. O grupo de otimalidade pode ser deduzido dependendo do problema.
- A palavra chave *type* se usa para definir a penalidade da restrição. Varias restrições podem ter a mesma penalidade.
- A palavra chave *forall* se usa para definir os índices que irão ser usados nessa restrição.

Limites

- Cada um dos índices deve ter um limite ou valor específico.

- Os limites são definidos em intervalos fechados ou abertos.
- Os valores dos limites podem ser: um número inteiro, uma variável ou outro índice. Além disso se pode somar variáveis ou índices com números. Por exemplo: num-1 ou i+1.
- Os índices também podem ter só valores inteiros. Por exemplo: m=7 ou m=num ou n=num-1.
- Pode-se definir também a desigualdade de dois índices. Por exemplo: i $\neq$ k.

Clausulas

- As cláusulas são escritas na Forma Normal Conjuntiva (FNC), isto é, como uma conjunção de disjunções.
- Se a cláusula está composta de um literal só, então não precisa de parênteses.
- Do contrario se terá parênteses para cada disjunto, assim:

  (DISJUNTO1) and (DISJUNTO2) and ... (DISJUNTOn)

  onde o DISJUNTO tem a seguinte forma:

  (LITERAL1 or LITERAL2 or ... LITERALn)

  e LITERAL pode ser ATOMO ou not ATOMO
- Nos índices da cláusula pode-se realizar soma ou subtração de inteiros.

Observações

- O nome dos índices é uma letra só.
- As restrições estão separadas por ponto e vírgula (;).

Na Figura 1.1.2 se mostram as restrições usadas para definir o TSP.

A.1.3 Definição das Penalidades

- Atribui-se a cada penalidade seu nível de prioridade correspondente utilizando números inteiros.
- Sempre deve-se começar com o nível 0.

Na Figura 1.1.3 se mostram as penalidades usadas no TSP.

```
integrity group type int1: forall[i,j]; 1<=i<=num,1<=j<=num: pos[i][j];
integrity group type wta: forall{i,j,l};
1<=i<=num,1<=j<=num,1<=l<=num;j!=l: (not pos[i][j] or not pos[i][l]);
optimality group type costo: forall[i,j,k];
1<=i<=num,1<=j<=4,1<=k<=num;i!=k: dist[i][k](pos[i][j] and pos[k][j+1]);
optimality group type costo: forall{i,j,k};
1<=i<=num,2<=j<=num,1<=k<=num;i!=k: dist[i][k](pos[i][j] and
pos[k][j-1]);
```

Figura 1.1.2: Definição das Restrições do TSP

```
penalty{
wta is level 2;
int1 is level 1;
costo is level 0;}
```

Figura 1.1.3: Definição das penalidades do TSP

A.2 Estrutura dos Arquivos Secundários

A.2.1 Arquivos das estruturas que representam neurônios

As matrizes definidas nas estruturas são conjuntos de neurônios, os quais têm algumas características que são inicializadas dentro do compilador. O objetivo destes arquivos é mudar esta configuração inicial. Os atributos de cada neurônio definidos no arquivo são:

(i) A posição do neurônio na matriz.

(ii) O estado ou valor do neurônio (0 ou 1).

(iii) O atributo fixo (1 quando se deseja que o valor do estado dado para esse neurônio não possa ser mudado e 0 no caso contrário).

(iv) A probabilidade de atualização (0 ou 1).

Estes 3 últimos atributos são inicializados com 0, 0 e 1 respectivamente.

Observações

- O nome do arquivo é igual ao nome da estrutura.

- Os três atributos são separados por hífens e a ordem é a detalhada acima.
- É uma linha para cada neurônio.
- Não é preciso mudar a configuração de todos os neurônios, só daqueles dos quais o usuário deseja.

A.2.2 Arquivos das Estruturas que não Representam Neurônios

Estes arquivos contêm os valores fixos a serem usados no processo de geração da função de energia. O formato de entrada é parecido ao usado nos arquivos de estruturas. A diferença é que depois das coordenadas se escreve o valor dado.

Apêndice B

Código do Analizador Léxico em Flex++

```
% name RestScanner
%define IOSTREAM
%define LEX_PARAMYY_RestParser_STYPE * val,
YY_RestParser_LTYPE * loc
%define MEMBERSpublic : intline, column;
%define CONSTRUCTOR_INIT : line(1), column(1)
%x comment

%header {
#include < sstream >
#include < stdio.h >
#include "parser.h"
% }

DIGIT09 [0-9]
DIGIT19 [1-9]

[fF][oO][rR][aA][lL][lL]   {
                           column += 6;
                           return RestParser::FORALL; }
[tT][yY][pP][eE]           {
                           column += 4;
```

```
                    return RestParser::TYPE; }

[pP][eE][nN][aA][lL][tT][yY]   {
                    column += 7;
                    return RestParser::PENALTY; }
[iI][sS]                   {
                    column += 2;
                    return RestParser::IS;}
[iI][nN][tT][eE][gG][rR][iI][tT][yY]   {
                    column += 9;
                    return RestParser::INTEGRITY;}
[oO][pP][tT][iI][mM][aA][lL][iI][tT][yY]  {
                    column += 10;
                    return RestParser::OPTIMALITY;}
[gG][rR][oO][uU][pP]     {
                    column += 5;
                    return RestParser::GROUP;}
[lL][eE][vV][eE][lL]     {
                    column += 5;
                    return RestParser::LEVEL;}
[rR][eE][aA][dD]          {
                    column += 4;
                    return RestParser::READ;}
[fF][rR][oO][mM]          {
                    column += 4;
                    return RestParser::FROM;}
[oO][rR]                  {
                    column += 2;
                    return RestParser::OR;}
[aA][nN][dD]              {
                    column += 3;
                    return RestParser::AND;}
[nN][oO][tT]              {
```

```
                    column += 3;
                    return RestParser::NOT;}

[iI][nN]           {
                    column += 2;
                    return RestParser::IN;}
"="                {
                    ++column;
                    return RestParser::EQUAL;}
"<"                {
                    ++column;
                    return RestParser::LESS ;}
"!"                {                    ++column;
                    return RestParser::NOTEQ ; }
"+"                {
                    ++column;
                    return RestParser::PLUS ; }
"-"                {
                    ++column;
                    return RestParser::MINUS ; }
","                {
                    ++column;
                    return RestParser::COMMA ; }
";"                {
                    ++column;
                    return RestParser::SEMICOLON ;}
":"                {
                    ++column;
                    return RestParser::COLON ; }
"("                {
                    ++column;
                    return RestParser::RPARENT ; }
")"                {
```

```
                        ++column;
                        return RestParser::LPARENT ;}


"{"                    {
                        ++column;
                        return RestParser::RCBRACKET ;}
"}"                    {
                        ++column;
                        return RestParser::LCBRACKET ; }
"["                    {
                        ++column;
                        return RestParser::RBRACKET ; }
"]"                    {
                        ++column;
                        return RestParser::LBRACKET ;}
[a-zA-Z][a-zA-Z0-9]+  {
                        column += strlen(yytext);
                        return RestParser::VARPROP ;}
[a-zA-Z][a-zA-Z0-9]+"."[t][x][t]  {
                        column += strlen(yytext);
                        return RestParser::FILE ;}
DIGIT09+               {
                        column += strlen(yytext);
                        return RestParser::INTEGER ; }
[a-zA-Z]               {
                        ++column;
                        return RestParser::INDEX ; }
" "                    {
                        ++column;}
\t                     {
                        column += 8;}
.                      {
```

```
                      ++column;
                      return RestParser::UNKNOWN; }
\ n                  {
                      column = 1;
                      ++line; }
[/][/][^\ n]*        {
                      column = 1;
                      ++line; }

"/*" BEGIN(comment);
<comment>[^* \n]*
<comment>"*"+[^* /\n]*
<comment>\n ++line;
<comment>"+"/" BEGIN(INITIAL);

«EOF»                { yyterminate(); }

%%
```

Apêndice C

Gramática da Linguagem do SATyrus

<start> ::= <structures> <constraints> <optimalities> <penalties>;

<structures> ::= <structure> SEMICOLON <structures1> ;

<structure> ::= VARPROP <structures2>;

<structures2> ::= RPARENT <dimensions> <dimensions2>
| READ FROM FILE
| <elements> EQUAL <value>
| EQUAL <operation3>;

<dimensions2> ::= LPARENT
| SEMICOLON <differences> LPARENT ;

<differences> ::= <difference> <differences1>;

<difference> ::= INDEX NOTEQ EQUAL INDEX;

<differences1> ::= //empty
| COMMA <differences>;

<operation3> ::= INTEGER <operation>
| VARPROP <operation> ;

<elements> ::= <element> <elements1>;

<element> ::= RBRACKET INTEGER LBRACKET;

<elements1> ::= //empty
| <element> <elements1>;

```
<value>          ::= INTEGER
                 | FLOAT;

<structures1>    ::= //empty
                 | <structure> SEMICOLON <structures1> ;

<dimensions>     ::= INTEGER <dimensions1>
                 | VARPROP <operation> <dimensions1>
                 | <interval> <dimensions1>;

<dimensions1>    ::= //empty
                 | COMMA <dimensions>;

<constraints>    ::= INTEGRITY GROUP <constraint> SEMICOLON
                 <constraints1>;

<constraints1>   ::= //empty
                 | INTEGRITY GROUP <constraint> SEMICOLON
                 <constraints1>;

<constraint>     ::= <type> <indexinis> <indexlimits> <indexdifs>
                 <clauses>;

<type>           ::= TYPE VARPROP COLON;

<indexinis>      ::= <indexini> <indexini1>;

<indexini>       ::= FORALL RCBRACKET <inindexes> LCBRACKET
                 | EXISTS RCBRACKET <inindexes> LCBRACKET ;

<indexini1>      ::= SEMICOLON
                 | EXISTS RCBRACKET <inindexes> LCBRACKET
                 | FORALL RCBRACKET <inindexes> LCBRACKET ;

<inindexes>      ::= INDEX <inindexes1>;

<inindexes1>     ::= //empty
                 | COMMA INDEX <inindexes1>;

<indexlimits>    ::= <interval> <intervals1>;

<intervals1>     ::= //empty
                 | COMMA <interval> <intervals1>;
```

```
<interval>      ::= INTEGER LESS <equal1> INDEX LESS <equal1> <limit>
                | INDEX <operation> <interval1>;
                | VARPROP <operation> <interval1>;

<interval1>     ::= LESS <equal1> INDEX LESS <equal1> <limit>
                | IN RCBRACKET list LCBRACKET;

<limit>         ::= INTEGER
                | INDEX <operation>
                | VARPROP <operation>;

<operation>     ::= //empty
                | sign <operation1>;
                | EQUAL <operation1>;

<operation1>    ::= INTEGER
                | INDEX
                | VARPROP;

<sign>          ::= PLUS
                | MINUS
                | TIMES;

<equal1>        ::= //empty
                | EQUAL;

<list>          ::= INTEGER COMMA INTEGER <list1>;

<list1>         ::= //empty
                | COMMA INTEGER <list1>;

<indexdifs>     ::= COLON
                | SEMICOLON indexdif <indexdifs1>;

<indexdif>      ::= INDEX <indexdifs2>;

<indexdifs2>    ::= NOTEQ EQUAL INDEX
                | EQUAL <indexdifs3>;

<indexdifs3>    ::= INTEGER
                | VARPROP
                | INDEX <operation>;

<indexdifs1>    ::= COLON
                | COMMA indexdif <indexdifs1>;
```

```
<clauses>          ::= <disjuncts>
                   | <coeficient> RPARENT <disjuncts> LPARENT ;

<coeficient>       ::= INTEGER
                   | <atom> ;

<disjuncts>        ::= <disjunct> <disjuncts>1;

<disjuncts1>       ::= //empty
                   | AND <disjunct> <disjuncts>1;

<disjunct>         ::= <literal>
                   | RPARENT <literal> literals LPARENT;

<literals>         ::= OR <literal> <literals1>;

<literals1>        ::= //empty
                   | OR <literal> <literals1>;

<literal>          ::= <atom>
                   | NOT <atom>;

<atom>             ::= VARPROP <elements2>;

<elements2>        ::= RBRACKET vindex LBRACKET <elements3>;

<elements3>        ::= //empty
                   | RBRACKET <vindex> LBRACKET <elements3>;

<vindex>           ::= INDEX <operation>;

<optimalities>     ::= //empty
                   |OPTIMALITY GROUP <constraint> SEMICOLON
                   <optimalities1>;

<optimalities1>    ::= //empty
                   | OPTIMALITY GROUP <constraint> SEMICOLON
                   <optimalities1>;

<penalties>        ::= PENALTY RCBRACKET <penalties1> LCBRACKET;

<penalties1>       ::= <penalty> SEMICOLON <penalties2>;
```

```
<penalties2>  ::= //empty
              | <penalty> SEMICOLON <penalties2>;

<penalty>     ::= VARPROP IS LEVEL INTEGER;
```

Apêndice D

Código Fonte da ARQ-PROP-II

```
num=4; /* num varia dependendo do número de linhas necessarias para a prova*/
iin(num);
proof(num,num,2);
cb(num);
res(num);
empty(num);
orig(num);
clcomp(num,num,2);
cbmap(num,num);
parent(num,num,2);
cancel(num,num);

//RESTRIÇÕES DAS ESTRUTURAS IN e PROOF
integrity group type int1: forall{i}; 1<=i<=num: (not iin[i] or cb[i] or res[i]);
integrity group type int0: forall{i,j,k}; 1<=i<=num,1<=j<=num,1<=k<=2: (not iin[i] or empty[i]
or parent[i][j][k]);
integrity group type int1: forall{i,j,k}; 1<=i<=num,1<=j<=num,1<=k<=2: (not iin[i] or not
empty[i] or not parent[i][j][k]);

//RESTRIÇÕES DA CLÁUSULA
integrity group type int0: forall{i,j}; 1<=i<=num,1<=j<=num: (not cb[i] or cbmap[i][j]);
integrity group type int1: forall{i,j}; 1<=i<=num,1<=j<=num: (not orig[j] or not cbmap[i][j] or
cb[i]);
```

//RESTRIÇÕES DA SINTAXE DA CLÁUSULA

```
integrity group type int1: forall{i,j,k,s}; 1<=i<=num,1<=j<=num,1<=k<=num,1<=s<=2: (not cbmap[i][j] or not clcomp[j][k][s] or proof[i][k][s]);
integrity group type int1: forall{i,j,k,s}; 1<=i<=num,1<=j<=num,1<=k<=num,1<=s<=2: (not cbmap[i][j] or clcomp[j][k][s] or not proof[i][k][s]);
```

//RESTRIÇÕES DOS PASSOS DE RESOLUÇÃO

```
integrity group type int1: forall{i,j,k,l,m}; 1<=i<=num,1<=j<=num,1<=k<=num;j!=k,l=1,m=2: (not res[i] or parent[j][i][l]) and (not res[i] or parent[k][i][m]);
integrity group type int0: forall{i,k};1<=i<=num,1<=k<=num: (not res[i] or cancel[i][k]);
```

//RESTRIÇÕES DA ESTRUTURA PARENT

```
integrity group type int1: forall{i,j,k,l,m,n,s,t};1<=i<=num,1<=j<=num,1<=k<=num,1<=l<=num, 1<=m<=2, 1<=n<=2;j!=k,m!=n,s=1,t=2:(not parent[j][i][s] or not parent[k][i][t] or not proof[j][l][m] or not proof[k][l][n] or cancel[i][l]);
integrity group type int1:forall{i,j,k};1<=i<=num,1<=j<=num,1<=k<=2:(res[i] or not parent[j][i][k]);
integrity group type int1:forall{i,j,k};1<=i<=num,1<=j<=num,1<=k<=2:(not parent[j][i][k] or iin[i]);
integrity group type int1:forall{i,j,k};1<=i<=num,1<=j<=num,1<=k<=2:(not parent[j][i][k] or iin[j]);
```

//RESTRIÇÕES DA COMPOSIÇÃO DO RESOLVENTE

```
integrity group type int1: forall{i,j,k,l,s};1<=i<=num,1<=j<=num,1<=k<=num,1<=l<=2,1<=s<=2; j!=i:(not parent[j][i][l] or not proof[j][k][s] or cancel[i][k] or proof[i][k][s]);
integrity group type int1: forall{i,j,k,l,m,n,p};1<=i<=num,1<=j<=num,1<=k<=num,1<=l<=num, 1<=p<=2; j!=k,k!=i,j!=i,m=1,n=2:(not parent[j][i][m] or not parent[k][i][n] or proof[j][l][p] or proof[k][l][p] or not proof[i][l][p]);
integrity group type int1: forall{i,j,s};1<=i<=num,1<=j<=num,1<=s<=2:(not cancel[i][j] or not proof[i][j][s]);
```

//RESTRIÇÕES DA CLÁUSULA VAZIA

```
integrity group type int1: forall{i,j,s};1<=i<=num,1<=j<=num,1<=s<=2:(not empty[i] or not
proof[i][j][s]);
integrity group type int1: forall{i};1<=i<=num:(not empty[i] or res[i]);

//RESTRIÇÕES WTA
integrity group type wta:forall{i,j,s,k};1<=i<=num,1<=j<=num,1<=s<=2,1<=k<=2;s!=k:(not
proof[i][j][s] or not proof[i][j][k]);
integrity group type wta: forall{i};1<=i<=num: (not cb[i] or not res[i]);
integrity group type wta: forall{i,j,k};1<=i<=num,1<=j<=num,1<=k<=num;j!=k:(not cbmap[i][j]
or not cbmap[i][k]);
integrity group type wta:forall{i,j,k,l};1<=i<=num,1<=j<=num,1<=k<=num,1<=l<=2;k!=j:(not
parent[j][i][l] or not parent[k][i][l]);
integrity group type wta:forall{i,j,n,k,l};1<=i<=num,1<=j<=num,1<=n<=num,1<=k<=2,1<=l<=2;
i!=n:(not parent[j][i][k] or not parent[j][n][l]);
integrity group type wta:forall{i,j,k,l}; 1<=i<=num,1<=j<=num,1<=k<=2,1<=l<=2;k!=l: (not
parent[j][i][k] or not parent[j][i][l]);
integrity group type wta: forall{i,j,k};1<=i<=num,1<=j<=num,1<=k<=num;j!=k:(not cancel[i][j]
or not cancel[i][k]);

penalty{
int0 is level 0;
int1 is level 1;
wta is level 2;
}
```

Apêndice E

Código Fonte dos Testes do Compilador

TESTE1

```
/******************************************
structure : VARPROP structures2
structures2 : EQUAL varoperation
varoperation : INTEGER varoperation1
varoperation1 : //empty
******************************************/
num=3;
pos(num,num);
integrity group type groupinteg1: forall {i,j}; 1<=i<=num,1<=j<=num:
pos[i][j];
penalty {
groupinteg1 is level 0;}
```

TESTE2

```
/******************************************
structure : VARPROP structures2
structures2 : EQUAL varoperation
varoperation : INTEGER varoperation1
varoperation1 : sign varoperation2
varoperation2 : INTEGER
sign : PLUS
```

***/

num=1;

num1=num+1;

pos(num1,num1);

integrity group type groupinteg1: **forall**{i,j}; 1<=i<=num1,1<=j<=num1: pos[i][j];

penalty{

groupinteg1 **is level** 0; }

TESTE3

/***

structure : VARPROP structures2

structures2 : EQUAL varoperation

varoperation : INTEGER varoperation1

varoperation1 : sign varoperation2

varoperation2 : VARPRO

sign : PLUS

***/

num=1;

num1=2;

num2=num+num1;

pos(num2,num2);

integrity group type groupinteg1: **forall**{i,j}; 1<=i<=num2,1<=j<=num2: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE4

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions : INTEGER dimensions1

dimensions1 : //empty | COMMA dimensions

```
dimensions2 : LPARENT
interval : INTEGER LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : INTEGER
*******************************************/
pos(2,2);
integrity group type groupinteg1: forall{i,j}; 1<=i<=2,1<=j<=2: pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE5

```
/*******************************************
structure : VARPROP structures2
structures2 : EQUAL varoperation
varoperation : INTEGER varoperation1
varoperation1 : sign varoperation2
varoperation2 : INTEGER
sign : MINUS
*******************************************/
num=4;
num1=num-2;
pos(num1,num1);
integrity group type groupinteg1: forall{i,j}; 1<=i<=num1,1<=j<=num1:
pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE6

```
/*******************************************
structure : VARPROP structures2
structures2 : EQUAL varoperation
varoperation : INTEGER varoperation1
varoperation1 : sign varoperation2
```

varoperation2 : VARPRO

sign : MINUS

***/

num=6;

num1=4;

num2=num-num1;

pos(num2,num2);

integrity group type groupinteg1: **forall**{i,j}; 1<=i<=num2,1<=j<=num2: pos[i][j];

penalty {

groupinteg1 **is level** 0;}

TESTE7

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : VARPROP varoperation1 dimensions1

varoperation1 : sign varoperation2

varoperation2 : INTEGER

dimensions1 : //empty | COMMA dimensions

sign : PLUS

interval : INTEGER LESS equal1 INDEX LESS equal1 limit

equal1 : EQUAL

limit : VARPROP intoperation

intoperation : sign operation1

operation1 : INTEGER

sign : PLUS | MINUS;

***/

num=1;

num1=4;

pos(num+1,num1-2);

integrity group type groupinteg1: **forall**{i,j}; 1<=i<=num1-2,1<=j<=num+1:

pos[i][j];
penalty{
groupinteg1 **is level** 0;}

TESTE8

*/***
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : VARPROP varoperation1 dimensions1
varoperation1 : sign varoperation2
varoperation2 : VARPROP
dimensions1 : //empty | COMMA dimensions
sign : PLUS
interval : INTEGER LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : VARPROP intoperation
intoperation : sign operation1
operation1 : VARPROP
sign : PLUS | MINUS;
***/*

num=1;
num1=1;
num2=3;
pos(num+num1,num2-num);
integrity group type groupinteg1: **forall**{i,j};
1<=i<=num+num1,1<=j<=num2-num: pos[i][j];
penalty{
groupinteg1 **is level** 0;}

TESTE9

*/***
structure : VARPROP structures2*

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INTEGER LESS equal1 INDEX LESS equal1 limit

equal1 : //empty

limit : INTEGER

***/

pos(0<i<3,0<j<3);

integrity group type groupinteg1: **forall**{i,j}; 0<i<3,0<j<3: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE10

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INTEGER LESS equal1 INDEX LESS equal1 limit

equal1 : //empty

limit : VARPROP intoperation

intoperation : //empty

***/

num=3;

pos(0<i<num,0<j<num);

integrity group type groupinteg1: **forall**{i,j}; 0<i<num,0<j<num: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE11

/***

```
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INTEGER LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : INTEGER
********************************************/
pos(1<=i<=2,1<=j<=2);
integrity group type groupinteg1: forall{i,j}; 1<=i<=2,1<=j<=2: pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE12

```
/********************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INTEGER LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : VARPROP intoperation
intoperation : //empty
********************************************/
num=2;
pos(1<=i<=num,1<=j<=num);
integrity group type groupinteg1: forall{i,j}; 1<=i<=num,1<=j<=num:
pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE13

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INTEGER LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : VARPROP intoperation
intoperation : sign operation1
sign : PLUS
operation1 : INTEGER
*******************************************/
num=1;
pos(1<=i<=num+1,1<=j<=num+1);
integrity group type groupinteg1: forall{i,j}; 1<=i<=num+1,1<=j<=num+1:
pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE14

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : PARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INTEGER LESS equal1 INDEX LESS equal1 limit
equal1 : //empty
limit : VARPROP intoperation
intoperation : sign operation1
sign : PLUS
```

operation1 : INTEGER

***/

num=1;

pos(0<i<num+2,0<j<num+2);

integrity group type groupinteg1: **forall**{i,j}; 0<i<num+2,0<j<num+2: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE15

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INTEGER LESS equal1 INDEX LESS equal1 limit

equal1 : //empty

limit : VARPROP intoperation

intoperation : sign operation1

sign : PLUS

operation1 : VARPROP

***/

num=1;

num1=2;

pos(0<i<num+num1,0<j<num+num1);

integrity group type groupinteg1: **forall**{i,j}; 0<i<num+num1,0<j<num+num1: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE16

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INTEGER LESS equal1 INDEX LESS equal1 limit

equal1 : EQUAL

limit : VARPROP intoperation

intoperation : sign operation1

sign : PLUS

operation1 : VARPROP

***/

num=1;

pos(1<=i<=num+num,1<=j<=num+num);

integrity group type groupinteg1: **forall**{i,j}; 1<=i<=num+num,1<=j<=num+num: pos[i][j];

penalty{
groupinteg1 **is level** 0;}

TESTE17

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : VARPROP intoperation interval1

intoperation : //empty

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : EQUAL

limit : INTEGER

***/

num=1;

pos(num<=i<=2,num<=j<=2);
integrity group type groupinteg1: **forall**{i,j}; num<=i<=2,num<=j<=2: pos[i][j];
penalty{
groupinteg1 **is level** 0;}

TESTE18

/***

structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : VARPROP intoperation interval1
intoperation : //empty
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : VARPROP

***/

num=1;
num2=2;
pos(num<=i<=num2,num<=j<=num2);
integrity group type groupinteg1: **forall**{i,j}; num<=i<=num2,num<=j<=num2: pos[i][j];
penalty{
groupinteg1 **is level** 0;}

TESTE19

/***

structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions
interval : VARPROP intoperation interval1
intoperation : //empty | sign operation1
sign : PLUS
operation1 : INTEGER
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : VARPROP intoperation
***/

num=1;
pos(num<=i<=num+2,num<=j<=num+2);
integrity group type groupinteg1: **forall**{i,j}; num<=i<=num+2,num<=j<=num+2: pos[i][j];
penalty{
groupinteg1 **is level** 0;}

TESTE20

/***
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : VARPROP intoperation interval1
intoperation : //empty | sign operation1
sign : PLUS
operation1 : VARPROP
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : VARPROP intoperation
***/

num=1;
num2=2;

```
pos(num<=i<=num+num2,num<=j<=num+num2);
integrity group type groupinteg1: forall{i,j};
num<=i<=num+num2,num<=j<=num+num2: pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE21

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : VARPROP intoperation interval1
intoperation : //empty
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : //empty
limit : INTEGER
*******************************************/
num=0;
pos(num<i<3,num<j<3);
integrity group type groupinteg1: forall{i,j}; num<i<3,num<j<3: pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE22

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : VARPROP intoperation interval1
```

intoperation : //empty

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : //empty

limit : VARPROP intoperation

**/

num=0;

num1=3;

pos(num<i<num1,num<j<num1);

integrity group type groupinteg1: **forall**{i,j}; num<i<num1,num<j<num1: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE23

/**

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : VARPROP intoperation interval1

intoperation : //empty | sign operation1

sign : PLUS

operation1 : INTEGER

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : //empty

limit : VARPROP intoperation

**/

num=0;

pos(num<i<num+3,num<j<num+3);

integrity group type groupinteg1: **forall**{i,j}; num<i<num+3,num<j<num+3: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE24

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : VARPROP intoperation interval1

intoperation : //empty | sign operation1

sign : PLUS

operation1 : VARPROP

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : //empty

limit : VARPROP intoperation

***/

num=0;

num1=3;

pos(num<i<num+num1,num<j<num+num1);

integrity group type groupinteg1: **forall**{i,j};

num<i<num+num1,num<j<num+num1: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE25

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INTEGER intoperation interval1

intoperation : //empty

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : //empty

limit : INDEX intoperation

***/

num1=3;

pos(0<i<num1,0<j<i);

integrity group type groupinteg1: **forall**{i,j}; 0<i<num1,0<j<i: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE26

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INTEGER intoperation interval1

intoperation : //empty | sign operation1

sign : PLUS

operation1 : INTEGER

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : //empty

limit : INDEX intoperation

***/

num1=3;

pos(0<i<num1,0<j<i+1);

integrity group type groupinteg1: **forall**{i,j}; 0<i<num1,0<j<i+1: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE27

```
/********************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INTEGER intoperation interval1
intoperation : //empty | sign operation1
sign : PLUS
operation1 : VARPROP
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : //empty
limit : INDEX intoperation
********************************************/
num1=3;
num2=1;
pos(0<i<num1,0<j<i+num2);
integrity group type groupinteg1: forall{i,j}; 0<i<num1,0<j<i-num2: pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE28

```
/********************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INTEGER intoperation interval1
intoperation : //empty
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
```

limit : INDEX intoperation
***/

num1=2;
pos(1<=i<=num1,1<=j<=i);
integrity group type groupinteg1: **forall**{i,j}; 1<=i<=num1,1<=j<=i:
pos[i][j];
penalty{
groupinteg1 **is level** 0;}

TESTE29

/***

structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INTEGER intoperation interval1
intoperation : //empty | sign operation1
sign : PLUS
operation1 : INTEGER
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : INDEX intoperation
***/

num1=2;
pos(1<=i<=num1,1<=j<=i+1);
integrity group type groupinteg1: **forall**{i,j}; 1<=i<=num1,1<=j<=i+1:
pos[i][j];
penalty{
groupinteg1 **is level** 0;}

TESTE30

/***

```
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation : //empty | sign operation1
sign : MINUS
operation1 : VARPROP
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : INDEX intoperation
*******************************************/
num1=2;
num2=1;
pos(1<=i<=num1,1<=j<=i-num2);
integrity group type groupinteg1: forall{i,j}; 1<=i<=num1,1<=j<=i-num2:
pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE31

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation : //empty
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : VARPROP intoperation;
```

```
*******************************************/
num1=2;
pos(1<=i<=num1,i<=j<=num1);
integrity group type groupinteg1: forall{i,j}; 1<=i<=num1,i<=j<=num1:
pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE32

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : VARPROP intoperation interval1
intoperation : //empty | sign operation1
sign : PLUS
operation1 : INTEGER
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : //empty
limit : INDEX intoperation
*******************************************/
num=0;
num1=3;
pos(num<i<num1,num<j<i+1);
integrity group type groupinteg1: forall{i,j}; num<i<num1,num<j<i+1:
pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE33

```
/*******************************************
```

```
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : VARPROP intoperation interval1
intoperation : //empty | sign operation1
sign : PLUS
operation1 : VARPROP
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : //empty
limit : INDEX intoperation
*******************************************/
num1=0;
num2=1;
pos(num1<i<3,num1<j<i+num2);
integrity group type groupinteg1: forall{i,j}; num1<i<3,num1<j<i+num2:
pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE34

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : VARPROP intoperation interval1
intoperation : //empty
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : INDEX intoperation
```

```
*******************************************/
num=1;
num1=2;
pos(num<=i<=num1,num<=j<=i);
integrity group type groupinteg1: forall{i,j}; num<=i<=num1,num<=j<=i:
pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE35

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : VARPROP intoperation interval1
intoperation : //empty | sign operation1
sign : PLUS
operation1 : INTEGER
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : INDEX intoperation
*******************************************/
num=1;
num1=2;
pos(num<=i<=num1,num<=j<=i+1);
integrity group type groupinteg1: forall{i,j};
num<=i<=num1,num<=j<=i+1: pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE36 */***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INTEGER intoperation interval1

intoperation : //empty | sign operation1

sign : MINUS

operation1 : VARPROP

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : EQUAL

limit : INDEX intoperation

***/*

num1=2;

num2=1;

pos(1<=i<=num1,1<=j<=i-num2);

integrity group type groupinteg1: **forall**{i,j}; 1<=i<=num1,1<=j<=i-num2: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE37

*/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INDEX intoperation interval1

intoperation : //empty

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : EQUAL

```
limit : VARPROP intoperation
*******************************************/
num1=2;
pos(1<=i<=num1,i<=j<=num1);
integrity group type groupinteg1: forall{i,j}; 1<=i<=num1,i<=j<=num1:
pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE38

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation : //empty | sign operation1
sign : PLUS
operation1 : INTEGER
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : VARPROP intoperation
*******************************************/
num1=2;
num2=1;
pos(1<=i<=num1,i<=j<=num2+1);
integrity group type groupinteg1: forall{i,j}; 1<=i<=num1,i<=j<=num2+1:
pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE39

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation : //empty | sign operation1
sign : PLUS
operation1 : INTEGER
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : VARPROP intoperation
*******************************************/
num1=3;
pos(1<=i<=num1, i+1<=j<=num1);
integrity group type groupinteg1: forall{i,j}; 1<=i<=num1,i+1<=j<=num1:
pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE40

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation : sign operation1
sign : PLUS
operation1 : INTEGER
```

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : EQUAL

limit : INTEGER

***/

pos(1<=i<=3, i+1<=j<=3);

integrity group type groupinteg1: **forall**{i,j}; 1<=i<=3,i+1<=j<=3: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE41

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INDEX intoperation interval1

intoperation : sign operation1

sign : PLUS

operation1 : VARPROP

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : EQUAL

limit : INTEGER

***/

num=1;

pos(1<=i<=3, i+num<=j<=3);

integrity group type groupinteg1: **forall**{i,j}; 1<=i<=3,i+num<=j<=3: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE42

/***

```
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation : //empty
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : VARPROP intoperation
*********************************************/
num=1;
num1=3;
pos(num<=i<=num1,i<=j<=num1);
integrity group type int1: forall{i,j}; num<i<num1,num<j<i: pos[i][j];
penalty{
int1 is level 0;}
```

TESTE43

```
/*********************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation ://empty | sign operation1
sign : PLUS
operation1 : VARPROP
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : VARPROP intoperation
```

***/
num=1;
num1=3;
pos(1<=i<=num1, i+num<=j<=num1);
integrity group type groupinteg1: **forall**{i,j};
1<=i<=num1,i+num<=j<=num1: pos[i][j];
penalty{
groupinteg1 **is level** 0;}

TESTE44

/***

structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation ://empty
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : //empty
limit : INTEGER

***/
pos(0<i<4, i<j<4);
integrity group type groupinteg1: **forall**{i,j}; 0<i<4,i<j<4: pos[i][j];
penalty{
groupinteg1 **is level** 0;}

TESTE45

/***

structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1

```
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation ://empty
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : //empty
limit : VARPROP intoperation
*******************************************/
num1=4;
pos(0<i<num1, i<j<num1);
integrity group type groupinteg1: forall{i,j}; 0<i<num1, i<j<num1: pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE46

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation ://empty | sign operation1
sign : PLUS
operation1 : INTEGER
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : //empty
limit : VARPROP intoperation
*******************************************/
num1=3;
pos(0<i<num1+1, i<j<num1+1);
integrity group type groupinteg1: forall{i,j}; 0<i<num1+1, i<j<num1+1:
pos[i][j];
penalty{
```

groupinteg1 **is level** 0;}

TESTE47

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INDEX intoperation interval1

intoperation : sign operation1

sign : PLUS

operation1 : INTEGER

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : //empty

limit : INTEGER

***/

pos(0<i<4, i+1<j<5);

integrity group type groupinteg1: **forall**{i,j}; 0<i<4, i+1<j<5: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE48

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INDEX intoperation interval1

intoperation ://empty | sign operation1

sign : PLUS

operation1 : INTEGER

```
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : //empty
limit : VARPROP intoperation
*******************************************/
num=4;
num1=5;
pos(0<i<num, i+1<j<num1);
integrity group type groupinteg1: forall{i,j}; 0<i<num, i+1<j<num1: pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE49

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation ://empty | sign operation1
sign : PLUS
operation1 : VARPROP
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : //empty
limit : VARPROP intoperation
*******************************************/
num=4;
num1=5;
num2=1;
pos(0<i<num, i+num2<j<num1);
integrity group type groupinteg1: forall{i,j}; 0<i<num, i+num2<j<num1:
pos[i][j];
penalty{
```

```
groupinteg1 is level 0;}
```

TESTE50

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation : sign operation1
sign : PLUS
operation1 : VARPROP
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : //empty
limit : INTEGER
*******************************************/
num2=1;
pos(0<i<4, i+num2<j<5);
integrity group type groupinteg1: forall{i,j}; 0<i<4, i+num2<j<5: pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE51

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation ://empty | sign operation1
sign : PLUS
```

operation1 : VARPROP

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : EQUAL

limit : VARPROP intoperation

***/

num=1;

num1=2;

pos(1<=i<=num1, i<=j<=num+num1);

integrity group type groupinteg1: **forall**{i,j}; 1<=i<=num1, i<=j<=num+num1: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE52

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INDEX intoperation interval1

intoperation : sign operation1

sign : PLUS

operation1 : INTEGER

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : EQUAL

limit : VARPROP intoperation

***/

num2=1;

pos(1<=i<=num2+2, i+1<=j<=num2+2);

integrity group type groupinteg1: **forall**{i,j}; 1<=i<=num2+2, i+1<=j<=num2+2: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE53

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INDEX intoperation interval1

intoperation : sign operation1

sign : PLUS

operation1 : INTEGER | VARPROP

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : EQUAL

limit : VARPROP intoperation

***/

num2=1;

num1=2;

pos(1<=i<=num2+num1, i+1<=j<=num2+num1);

integrity group type groupinteg1: **forall**{i,j}; 1<=i<=num2+num1, i+1<=j<=num2+num1: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE54

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INDEX intoperation interval1

```
intoperation : sign operation1
sign : PLUS
operation1 : VARPROP
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : VARPROP intoperation
*******************************************/
num2=1;
num1=2;
pos(1<=i<=num2+num1, i+num2<=j<=num2+num1);
integrity group type groupinteg1: forall{i,j}; 1<=i<=num2+num1,
i+num2<=j<=num2+num1: pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE55

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation ://empty | sign operation1
sign : PLUS
operation1 : VARPROP
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : //empty
limit : VARPROP intoperation
*******************************************/
num=1;
num1=2;
pos(0<i<num+num1, i-1<j<num+num1);
```

integrity group type groupinteg1: **forall**{i,j}; 0<i<num+num1, i-1<j<num+num1: pos[i][j];
penalty{
groupinteg1 **is level** 0;}

TESTE56

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation : sign operation1
sign : PLUS
operation1 : VARPROP | INTEGER
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : //empty
limit : VARPROP intoperation
*******************************************/
```

num=2;
num1=3;
pos(0<i<num1, i+1<j<num+num1);
integrity group type groupinteg1: **forall**{i,j}; 0<i<num1, i+1<j<num+num1: pos[i][j];
penalty{
groupinteg1 **is level** 0;}

TESTE57

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
```

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INDEX intoperation interval1

intoperation : sign operation1

sign : PLUS

operation1 : VARPROP

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : //empty

limit : VARPROP intoperation

**/

num=2;

num1=3;

num2=1;

pos(0<i<num1, i+num2<j<num+num1);

integrity group type groupinteg1: **forall**{i,j}; 0<i<num1, i+num2<j<num+num1: pos[i][j];

penalty{

groupinteg1 **is level** 0;}

TESTE58

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INDEX intoperation interval1

intoperation : sign operation1

sign : PLUS

operation1 : INTEGER

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : //empty

limit : VARPROP intoperation

```
*******************************************/
num=4;
num1=3;
pos(0<i<num, i+1<j<num1+2);
integrity group type groupinteg1: forall{i,j}; 0<i<num, i+1<j<num1+2:
pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE59

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation : sign operation1
sign : PLUS
operation1 : INTEGER | VARPROP
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : //empty
limit : VARPROP intoperation
*******************************************/
num1=4;
num=1;
pos(0<i<num1+1, i+num<j<num1+1);
integrity group type groupinteg1: forall{i,j}; 0<i<num1+1,
i+num<j<num1+1: pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE60

```
/********************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INDEX intoperation interval1
intoperation : sign operation1
sign : PLUS
operation1 : INTEGER | VARPROP
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : VARPROP intoperation
********************************************/
num=1;
num1=2;
pos(1<=i<=num1+1, i+num<=j<=num1+1);
integrity group type groupinteg1: forall{i,j}; 1<=i<=num1+1,
i+num<=j<=num1+1: pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE61

```
/********************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : SEMICOLON differences LPARENT
differences : difference differences1;
difference : INDEX NOTEQ EQUAL INDEX;
differences1 : //empty
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
```

```
interval : INTEGER interval1
intoperation : //empty
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : VARPROP intoperation
*******************************************/
num=2;
pos(1<=i<=num,1<=j<=num;j!=i);
integrity group type groupinteg1: forall{i,j}; 1<=i<=num,1<=j<=num;j!=i:
pos[i][j];
penalty{
groupinteg1 is level 0;}
```

TESTE62

```
/*******************************************
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : SEMICOLON differences LPARENT
differences : difference differences1;
difference : INDEX NOTEQ EQUAL INDEX;
differences1 : //empty
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INTEGER interval1
intoperation : //empty
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : //empty
limit : VARPROP intoperation
*******************************************/
num=3;
pos(0<i<num,0<j<num;j!=i);
integrity group type groupinteg1: forall{i,j}; 0<i<num,0<j<num;j!=i:
pos[i][j];
```

penalty{
groupinteg1 **is level** 0;}

TESTE63

*/**

structure : VARPROP structures2

structures2 : READ FROM FILE

clauses : coeficient RPARENT disjuncts LPARENT ;

disjuncts : disjunct disjuncts1;

disjuncts1 : //empty

disjunct : literal

literal : atom

atom : VARPROP elements2;

elements2 : RBRACKET vindex LBRACKET elements3;

elements3 : //empty

vindex : INDEX operation;

operation : //empty

**/*

num=2;
pos(num,num);
dist (num,num);
dist read from tsp3.txt;
integrity group type groupinteg1: **forall**{i,j}; 1<=i<=num,1<=j<=num: pos[i][j];
optimality group type costo: **forall**{i,j}; 1<=i<=num,1<=j<=num: dist[i][j](pos[i][j]);
penalty{
groupinteg1 **is level** 1;
costo **is level** 0;}

TESTE64

*/**

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : VARPROP varoperation1 dimensions1
varoperation1 : //empty
dimensions1 : //empty | COMMA dimensions
***/

num1=1;
num=2;
pos(num1,num,num);
integrity group type groupinteg1: **forall**{i,j,k}; 1<=i<=1,1<=j<=num, 1<=k<=num: pos[i][j][k];
penalty{
groupinteg1 **is level** 0;}

TESTE65

/***
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : INTEGER dimensions1
dimensions1 : //empty | COMMA dimensions
***/

pos(1,2,2);
integrity group type groupinteg1: **forall**{i,j,k}; 1<=i<=1,1<=j<=2, 1<=k<=2: pos[i][j][k];
penalty{
groupinteg1 **is level** 0; }

TESTE66

/***
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INTEGER intoperation interval1

intoperation : //empty

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : EQUAL

limit : VARPROP intoperation

***/

num=2;

pos(1<=i<=1,1<=j<=num,1<=k<=num);

integrity group type groupinteg1: **forall**{i,j,k}; 1<=i<=1,1<=j<=num, 1<=k<=num: pos[i][j][k];

penalty{

groupinteg1 **is level** 0; }

TESTE67

/***

structure : VARPROP structures2

structures2 : RPARENT dimensions dimensions2

dimensions2 : LPARENT

dimensions : interval dimensions1

dimensions1 : //empty | COMMA dimensions

interval : INTEGER intoperation interval1

intoperation : //empty | sign operation1

sign : MINUS

operation1 : INTEGER

interval1 : LESS equal1 INDEX LESS equal1 limit

equal1 : EQUAL

limit : VARPROP intoperation | INDEX intoperation

***/

num=3;

pos(1<=i<=num,1<=j<=i-1,1<=k<=2);

integrity group type groupinteg1: **forall**{i,j,k}; 1<=i<=num,

1<=j<=i-1,1<=k<=2: pos[i][j][k];
penalty{
groupinteg1 **is level** 0; }

TESTE68

/***
structure : VARPROP structures2
structures2 : RPARENT dimensions dimensions2
dimensions2 : LPARENT
dimensions : interval dimensions1
dimensions1 : //empty | COMMA dimensions
interval : INTEGER intoperation interval1 | INDEX intoperation interval1
intoperation : //empty | sign operation1
sign : PLUS
operation1 : INTEGER
interval1 : LESS equal1 INDEX LESS equal1 limit
equal1 : EQUAL
limit : VARPROP intoperation | INDEX intoperation
***/

num=3;
pos(1<=i<=num,i+1<=j<=num,1<=k<=2);
integrity group type groupinteg1: **forall**{i,j,k}; 1<=i<=num,i+1<=j<=num,
1<=k<=2: pos[i][j][k];
penalty{
groupinteg1 **is level** 0; }

TESTE69

/***
clauses : disjuncts
disjuncts : disjunct disjuncts1
disjunct : RPARENT literal literals LPARENT
disjuncts1 : //empty

literal : atom / NOT atom
literals : OR literal literals1
literals1 : //empty
atom : VARPROP elements2
elements2 : RBRACKET vindex LBRACKET elements3
elements3 : //empty / RBRACKET vindex LBRACKET elements3
vindex : INDEX operation
operation : //empty
**/

num=2;
iin(num);
cb(num);
integrity group type groupinteg1: **forall**{i}; 1<=i<=num: (not iin[i] or cb[i]);
penalty{
groupinteg1 **is level** 0; }

TESTE70

/**
clauses : disjuncts
disjuncts : disjunct disjuncts1
disjunct : RPARENT literal literals LPARENT
disjuncts1 : //empty / AND disjunct disjuncts1
literal : atom / NOT atom
literals : OR literal literals1
literals1 : //empty
atom : VARPROP elements2
elements2 : RBRACKET vindex LBRACKET elements3
elements3 : //empty / RBRACKET vindex LBRACKET elements3
vindex : INDEX operation
operation : //empty
**/

num=2;
iin(num);

cb(num);
integrity group type groupinteg1: **forall**{i}; 1<=i<=num: (not iin[i] or cb[i]) and (not cb[i] or iin[i]);
penalty{
groupinteg1 **is level** 0; }

TESTE71

num=2;
iin(num);
cb(num);
integrity group type groupinteg1: **forall**{i}; 1<=i<=num: (not iin[i] or cb[i]);
integrity group type groupinteg1: **forall**{i}; 1<=i<=num: (not cb[i] or iin[i]);
penalty{
groupinteg1 **is level** 0; }

Apêndice F

Resultados dos Testes do Compilador

TESTES 1-24

A função de energia gerada foi: $4 - 1 * x(1) - 1 * x(2) - 1 * x(3) - 1 * x(4)$.

Onde:

$x(1) \rightarrow$ pos[1][1]

$x(2) \rightarrow$ pos[2][1]

$x(3) \rightarrow$ pos[1][2]

$x(4) \rightarrow$ pos[2][2]

A Tabela F.1 mostra todas as combinações das variáveis $x(1)$, $x(2)$, $x(3)$ e $x(4)$. A coluna x representa o valor atribuído para cada combinação no gráfico e a coluna $f(x)$ o valor da função. Na Figura 6.0.1 foi representado o espaço de busca da nossa função de energia gerada. No canto superior direito se mostra o gráfico da configuração da matriz *pos* no mínimo global.

TESTES: 25, 30, 31 e 36

A função de energia gerada foi: $3 - 1 * x(1) - 1 * x(2) - 1 * x(3)$.

Onde:

$x(1) \rightarrow$ pos[2][1]

$x(2) \rightarrow$ pos[3][1]

$x1$	$x2$	$x3$	$x4$	x	$f(x)$
0	0	0	0	1	4
0	0	0	1	2	3
0	0	1	0	3	3
0	0	1	1	4	2
0	1	0	0	5	3
0	1	0	1	6	2
0	1	1	0	7	2
0	1	1	1	8	1
1	0	0	0	9	3
1	0	0	1	10	2
1	0	1	0	11	2
1	0	1	1	12	1
1	1	0	0	13	2
1	1	0	1	14	1
1	1	1	0	15	1
1	1	1	1	16	0

Tabela F.1: Tabela de Combinações para 4 Variáveis.

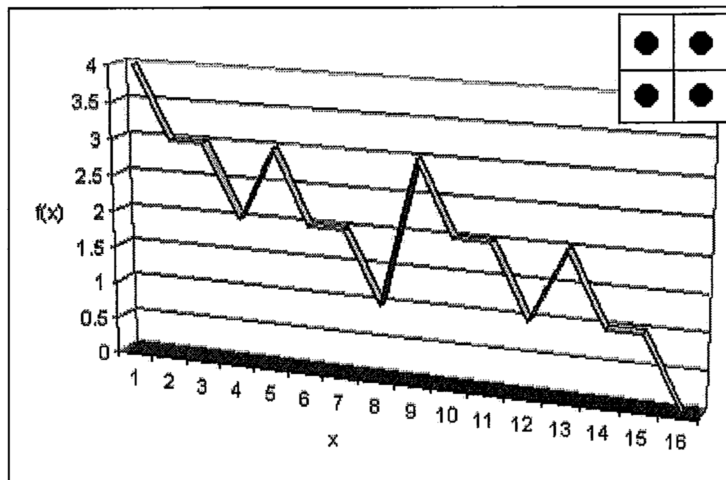


Figura 6.0.1: Espaço de Busca da Função de Energia (TESTE1 ate TESTE24).

$x1$	$x2$	$x3$	x	$f(x)$
0	0	0	1	3
0	0	1	2	2
0	1	0	3	2
0	1	1	4	1
1	0	0	5	2
1	0	1	6	1
1	1	0	7	1
1	1	1	8	0

Tabela F.2: Tabela de Combinações para 3 Variáveis.

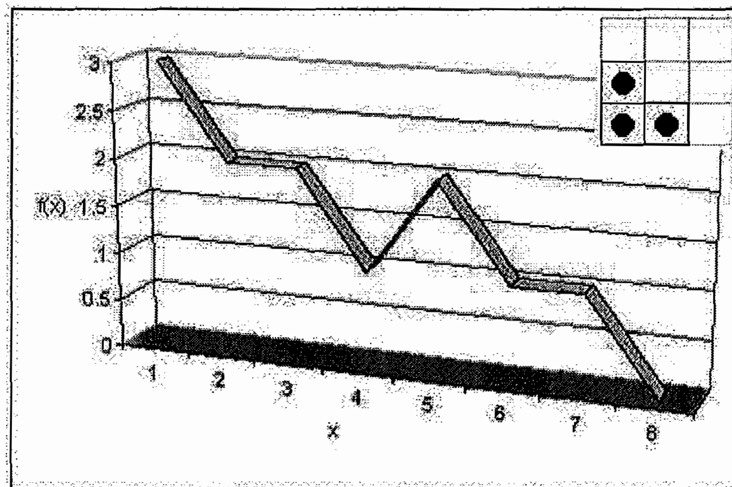


Figura 6.0.2: Espaço de Busca da Função de Energia (TESTES 25, 30, 31 e 36).

$x(3) \rightarrow$ pos[3][2]

A Tabela F.2 mostra todas as combinações das variáveis $x(1)$, $x(2)$ e $x(3)$. Na Figura 6.0.2 foi representado a o espaço de busca da nossa função de energia gerada. No canto superior direito se mostra o gráfico da configuração da matriz *pos* no mínimo global.

TESTES 26, 27, 28, 32, 33 e 34

A função de energia gerada foi: $3 - 1 * x(1) - 1 * x(2) - 1 * x(3)$.

Onde:

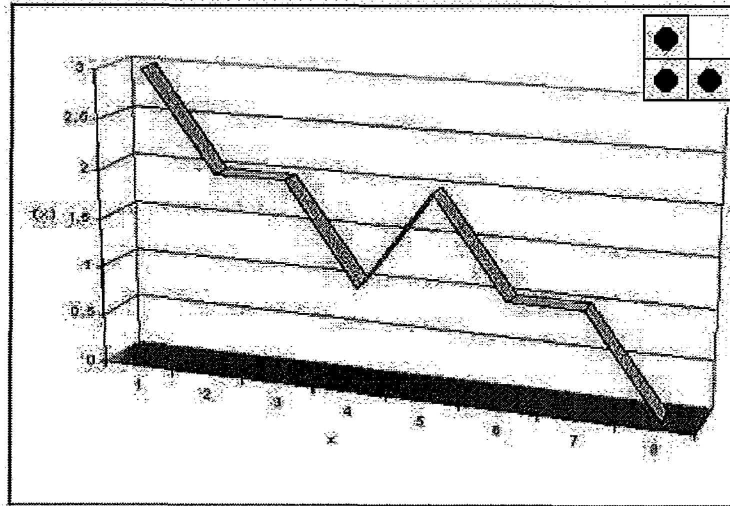


Figura 6.0.3: Espaço de Busca da Função de Energia (TESTES 26, 27, 28 e 32-34).

$x(1) \rightarrow$ pos[1][1]

$x(2) \rightarrow$ pos[2][1]

$x(3) \rightarrow$ pos[2][2]

A Tabela F.2 mostra todas as combinações das variáveis $x(1)$, $x(2)$ e $x(3)$. Na Figura 6.0.3 foi graficado a o espaço de busca da nossa função de energia gerada. Na esquina superior direita se mostra o gráfico da configuração da matriz *pos* no mínimo global.

TESTES: 29 e 35

A função de energia gerada foi: $5-1*x(1)-1*x(2)-1*x(3)-1*x(4)-1*x(5)$. Onde:

$x(1) \rightarrow$ pos[1][1]

$x(2) \rightarrow$ pos[1][2]

$x(3) \rightarrow$ pos[2][1]

$x(4) \rightarrow$ pos[2][2]

$x(5) \rightarrow$ pos[2][3]

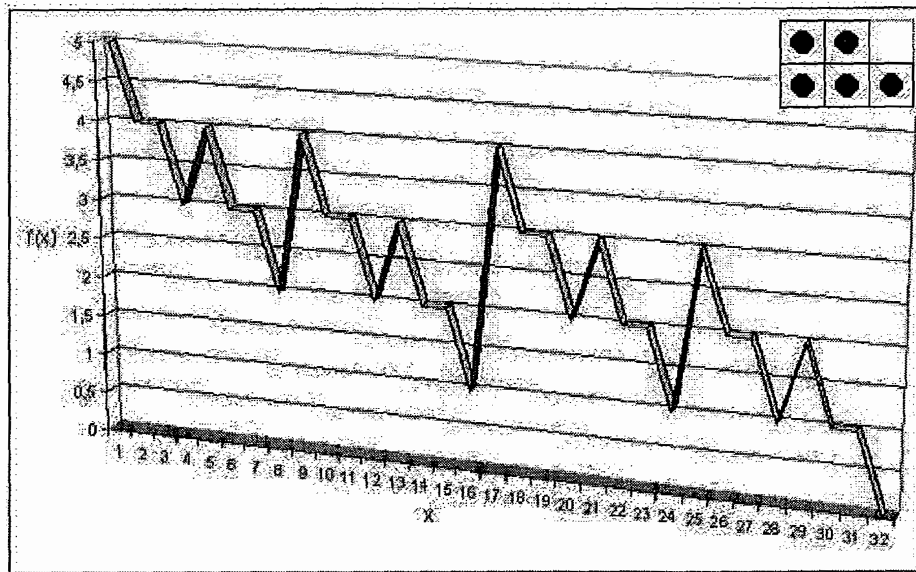


Figura 6.0.4: Espaço de Busca da Função de Energia (TESTES 29 e 35).

A Tabela F.3 mostra todas as combinações das variáveis $x(1)$, $x(2)$, $x(3)$, $x(4)$ e $x(5)$. Na Figura 6.0.4 foi graficado a o espaço de busca da nossa função de energia gerada. Na esquina superior direita se mostra o gráfico da configuração da matriz *pos* no mínimo global.

TESTES: 37, 38, 42 e 55

A função de energia gerada foi: $3 - 1 * x(1) - 1 * x(2) - 1 * x(3)$.

Onde:

$x(1) \rightarrow$ pos[1][1]

$x(2) \rightarrow$ pos[1][2]

$x(3) \rightarrow$ pos[2][2]

A Tabela F.2 mostra todas as combinações das variáveis $x(1)$, $x(2)$ e $x(3)$. Na Figura 6.0.5 foi graficado a o espaço de busca da nossa função de energia gerada. Na esquina superior direita se mostra o gráfico da configuração da matriz *pos* no mínimo global.

$x1$	$x2$	$x3$	$x4$	$x5$	x	$f(x)$
0	0	0	0	0	1	5
0	0	0	0	1	2	4
0	0	0	1	0	3	4
0	0	0	1	1	4	3
0	0	1	0	0	5	4
0	0	1	0	1	6	3
0	0	1	1	0	7	3
0	0	1	1	1	8	2
0	1	0	0	0	9	4
0	1	0	0	1	10	3
0	1	0	1	0	11	3
0	1	0	1	1	12	2
0	1	1	0	0	13	3
0	1	1	0	1	14	2
0	1	1	1	0	15	2
0	1	1	1	1	16	1
1	0	0	0	0	17	4
1	0	0	0	1	18	3
1	0	0	1	0	19	3
1	0	0	1	1	20	2
1	0	1	0	0	21	3
1	0	1	0	1	22	2
1	0	1	1	0	23	2
1	0	1	1	1	24	1
1	1	0	0	0	25	3
1	1	0	0	1	26	2
1	1	0	1	0	27	2
1	1	0	1	1	28	1
1	1	1	0	0	29	2
1	1	1	0	1	30	1
1	1	1	1	0	31	1
1	1	1	1	1	32	0

Tabela F.3: Tabela de Combinações para 5 Variáveis.

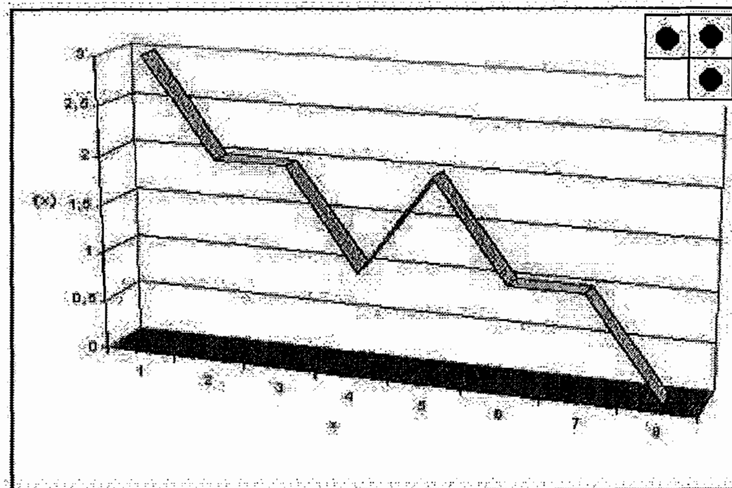


Figura 6.0.5: Espaço de Busca da Função de Energia (TESTES 37, 38, 42 e 55).

TESTES: 39, 40, 41, 43-46 e 52-54

A função de energia gerada foi: $3 - 1 * x(1) - 1 * x(2) - 1 * x(3)$.

Onde:

$x(1) \to$ pos[1][2]

$x(2) \to$ pos[1][3]

$x(3) \to$ pos[2][3]

A Tabela F.2 mostra todas as combinações das variáveis $x(1)$, $x(2)$ e $x(3)$. Na Figura 6.0.6 foi graficado a o espaço de busca da nossa função de energia gerada. Na esquina superior direita se mostra o gráfico da configuração da matriz *pos* no mínimo global.

TESTES: 47, 48-50, 56-59

A função de energia gerada foi: $3 - 1 * x(1) - 1 * x(2) - 1 * x(3)$.

Onde:

$x(1) \to$ pos[1][3]

$x(2) \to$ pos[1][4]

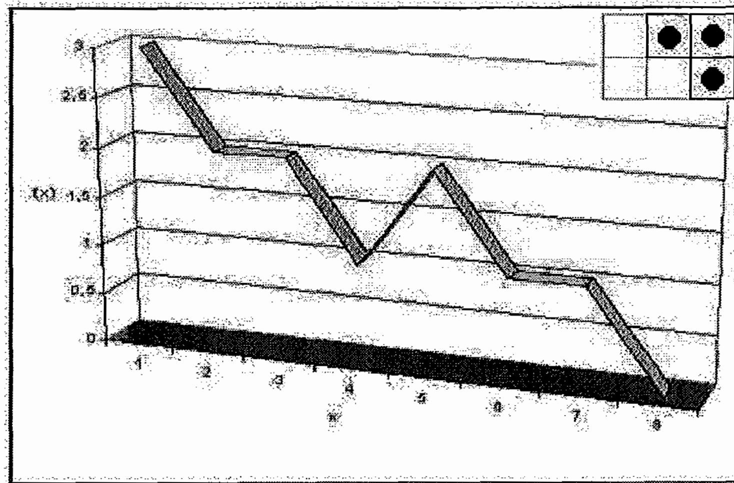


Figura 6.0.6: Espaço de Busca da Função de Energia (TESTES 39, 40, 41, 43-46 e 52-54).

$x(3) \rightarrow$ pos[2][4]

A Tabela F.2 mostra todas as combinações das variáveis $x(1)$, $x(2)$ e $x(3)$. Na Figura 6.0.7 foi graficado a o espaço de busca da nossa função de energia gerada. Na esquina superior direita se mostra o gráfico da configuração da matriz *pos* no mínimo global.

TESTE51

A função de energia gerada foi: $5 - 1 * x(1) - 1 * x(2) - 1 * x(3) - 1 * x(4) - 1 * x(5)$. Onde:

$x(1) \rightarrow$ pos[1][1]
$x(2) \rightarrow$ pos[1][2]
$x(3) \rightarrow$ pos[1][3]
$x(4) \rightarrow$ pos[2][2]
$x(5) \rightarrow$ pos[2][3]

A Tabela F.3 mostra todas as combinações das variáveis $x(1)$, $x(2)$, $x(3)$, $x(4)$ e $x(5)$. Na Figura 6.0.8 foi graficado a o espaço de busca da nossa função de energia gerada. Na esquina superior direita se mostra o gráfico da configuração da

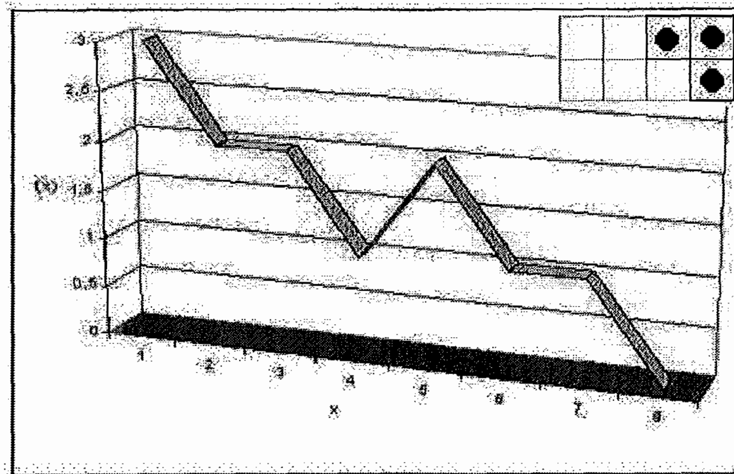


Figura 6.0.7: Espaço de Busca da Função de Energia (TESTES 47, 48-50, 56-59).

matriz *pos* no mínimo global.

TESTES 61 e 62

A função de energia gerada foi: $2 - 1 * x(1) - 1 * x(2)$.

Onde:

$x(1) \rightarrow$ pos[1][2]
$x(2) \rightarrow$ pos[2][1]

A Tabela F.4 mostra todas as combinações das variáveis $x(1)$ e $x(2)$. Na Figura 6.0.9 foi graficado a o espaço de busca da nossa função de energia gerada. Na esquina superior direita se mostra o gráfico da configuração da matriz *pos* no mínimo global.

TESTE 63

A função de energia gerada foi: $136 - 34 * x(1) - 34 * x(2) - 34 * x(3) - 34 * x(4) + 32 * x(1) + 40 * x(2) + 16 * x(3) + 64 * x(4)$.

Onde:

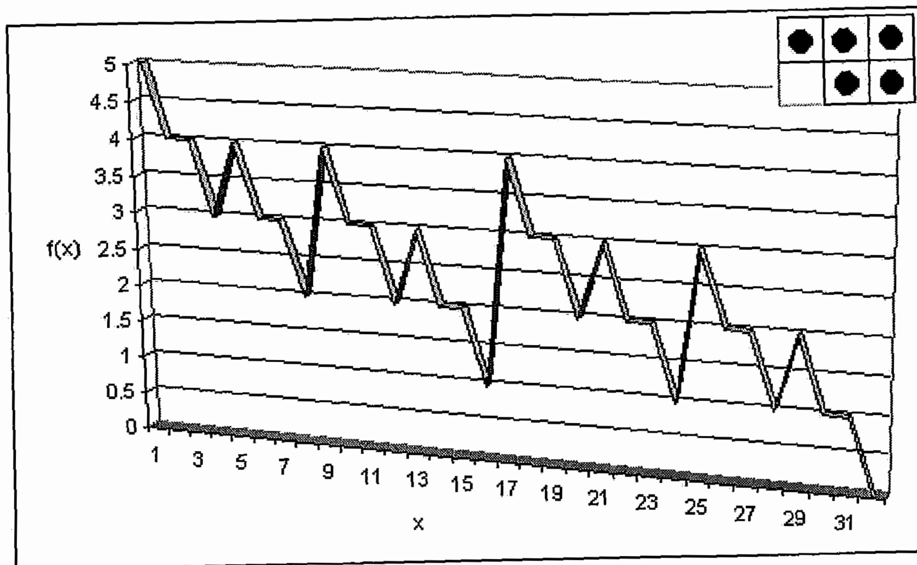


Figura 6.0.8: Espaço de Busca da Função de Energia (TESTE 51).

$x1$	$x2$	x	$f(x)$
0	0	1	2
0	1	2	1
1	0	3	1
1	1	4	0

Tabela F.4: Tabela de Combinações para 2 Variáveis.

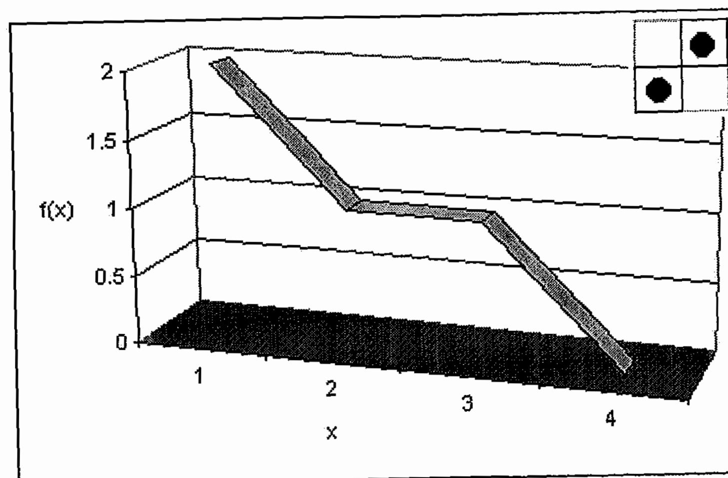


Figura 6.0.9: Espaço de Busca da Função de Energia (TESTES 61 e 62).

$x1$	$x2$	$x3$	$x4$	x	$f(x)$
0	0	0	0	1	136
0	0	0	1	2	166
0	0	1	0	3	118
0	0	1	1	4	148
0	1	0	0	5	142
0	1	0	1	6	172
0	1	1	0	7	124
0	1	1	1	8	154
1	0	0	0	9	198
1	0	0	1	10	228
1	0	1	0	11	180
1	0	1	1	12	210
1	1	0	0	13	204
1	1	0	1	14	234
1	1	1	0	15	186
1	1	1	1	16	216

Tabela F.5: Tabela das Combinações para o TESTE 63.

$x(1) \rightarrow$ pos[1][1]
$x(2) \rightarrow$ pos[2][1]
$x(3) \rightarrow$ pos[1][2]
$x(4) \rightarrow$ pos[2][2]

A Tabela F.5 mostra todas as combinações das variáveis $x(1)$, $x(2)$, $x(3)$ e $x(4)$. Na Figura 6.0.10 foi graficado a o espaço de busca da nossa função de energia gerada, e também as distâncias entre nós.

TESTES: 64-66

A função de energia gerada foi: $4 - 1 * x(1) - 1 * x(2) - 1 * x(3) - 1 * x(4)$. Onde:

$x(1) \rightarrow$ pos[1][1][1]
$x(2) \rightarrow$ pos[1][2][1]
$x(3) \rightarrow$ pos[1][1][2]
$x(4) \rightarrow$ pos[1][2][2]

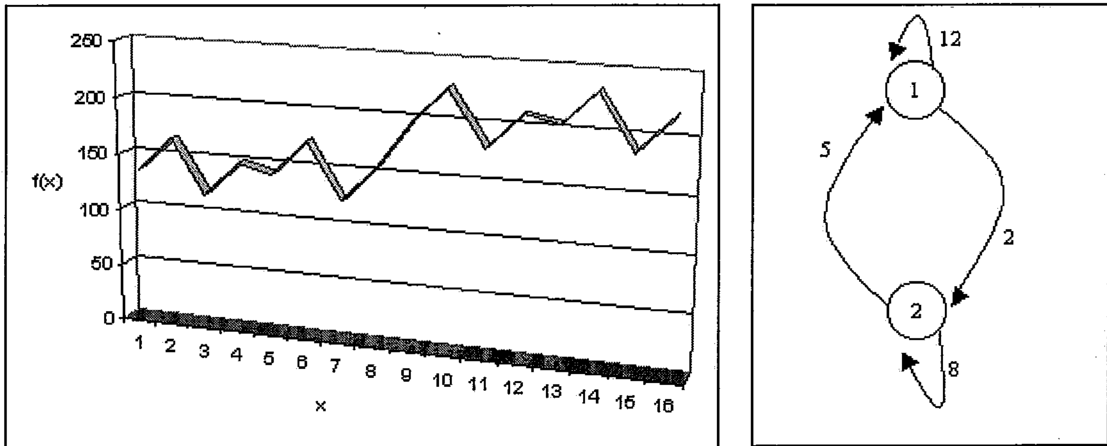


Figura 6.0.10: Espaço de Busca da Função de Energia (TESTE 63).

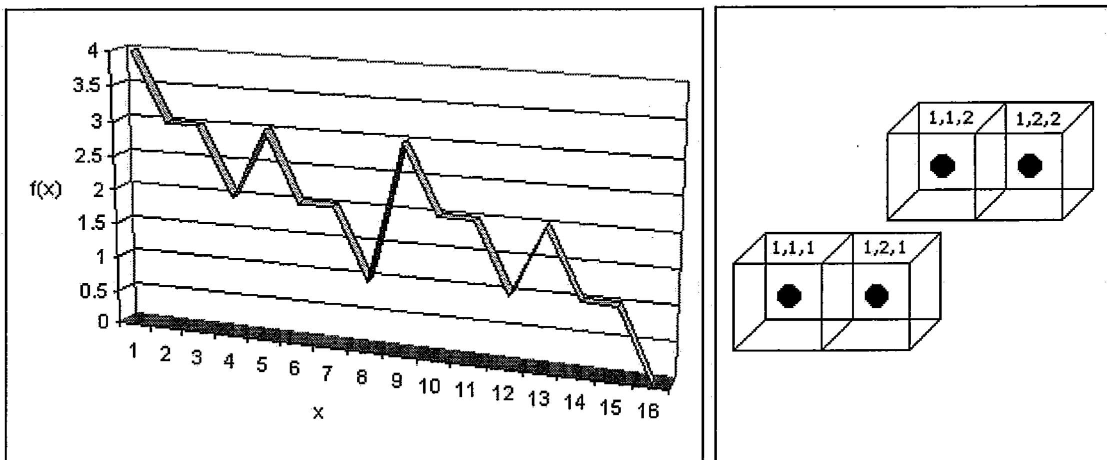


Figura 6.0.11: Espaço de Busca da Função de Energia (TESTES 64-66).

A Tabela ?? mostra todas as combinações das variáveis $x(1)$, $x(2)$, $x(3)$ e $x(4)$. Na Figura 6.0.11 foi graficado a o espaço de busca da nossa função de energia gerada e do lado se mostra o gráfico da configuração da matriz *pos* no mínimo global.

TESTE67

A função de energia gerada foi: $6 - 1 * x(1) - 1 * x(2) - 1 * x(3) - 1 * x(4) - 1 * x(5) - 1 * x(6)$.

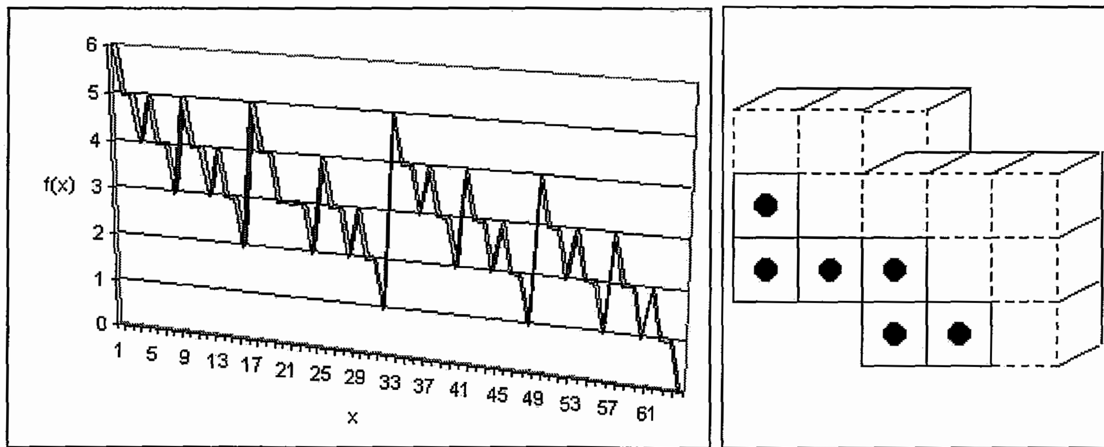


Figura 6.0.12: Espaço de Busca da Função de Energia (TESTE 67).

Onde:

$x(1) \rightarrow$ pos[2][1][1]
$x(2) \rightarrow$ pos[3][1][1]
$x(3) \rightarrow$ pos[3][2][1]
$x(4) \rightarrow$ pos[2][1][2]
$x(5) \rightarrow$ pos[3][1][2]
$x(6) \rightarrow$ pos[3][2][2]

As Tabelas F.6 e F.7 mostram todas as combinações das variáveis $x(1)$, $x(2)$, $x(3)$, $x(4)$, $x(5)$ e $x(6)$. Na Figura 6.0.12 foi graficado a o espaço de busca da nossa função de energia gerada e do lado se mostra o gráfico da configuração da matriz *pos* no mínimo global.

TESTE68

A função de energia gerada foi: $6 - 1 * x(1) - 1 * x(2) - 1 * x(3) - 1 * x(4) - 1 * x(5) - 1 * x(6)$.

Onde:

$x(1) \rightarrow$ pos[1][2][1]
$x(2) \rightarrow$ pos[1][3][1]
$x(3) \rightarrow$ pos[2][3][1]

$x1$	$x2$	$x3$	$x4$	$x5$	$x6$	x	$f(x)$
0	0	0	0	0	0	1	6
0	0	0	0	0	1	2	5
0	0	0	0	1	0	3	5
0	0	0	0	1	1	4	4
0	0	0	1	0	0	5	5
0	0	0	1	0	1	6	4
0	0	0	1	1	0	7	4
0	0	0	1	1	1	8	3
0	0	1	0	0	0	9	5
0	0	1	0	0	1	10	4
0	0	1	0	1	0	11	4
0	0	1	0	1	1	12	3
0	0	1	1	0	0	13	4
0	0	1	1	0	1	14	3
0	0	1	1	1	0	15	3
0	0	1	1	1	1	16	2
0	1	0	0	0	0	17	5
0	1	0	0	0	1	18	4
0	1	0	0	1	0	19	4
0	1	0	0	1	1	20	3
0	1	0	1	0	0	21	3
0	1	0	1	0	1	22	3
0	1	0	1	1	0	23	3
0	1	0	1	1	1	24	2
0	1	1	0	0	0	25	4
0	1	1	0	0	1	26	3
0	1	1	0	1	0	27	3
0	1	1	0	1	1	28	2
0	1	1	1	0	0	29	3
0	1	1	1	0	1	30	2
0	1	1	1	1	0	31	2
0	1	1	1	1	1	32	1
1	0	0	0	0	0	33	5
1	0	0	0	0	1	34	4
1	0	0	0	1	0	35	4
1	0	0	0	1	1	36	3
1	0	0	1	0	0	37	4
1	0	0	1	0	1	38	3
1	0	0	1	1	0	39	3

Tabela F.6: Tabela de Combinações para 6 Variáveis (parte 1).

$x1$	$x2$	$x3$	$x4$	$x5$	$x6$	x	$f(x)$
1	0	0	1	1	1	40	2
1	0	1	0	0	0	41	4
1	0	1	0	0	1	42	3
1	0	1	0	1	0	43	3
1	0	1	0	1	1	44	2
1	0	1	1	0	0	45	3
1	0	1	1	0	1	46	2
1	0	1	1	1	0	47	2
1	0	1	1	1	1	48	1
1	1	0	0	0	0	49	4
1	1	0	0	0	1	50	3
1	1	0	0	1	0	51	3
1	1	0	0	1	1	52	2
1	1	0	1	0	0	53	3
1	1	0	1	0	1	54	2
1	1	0	1	1	0	55	2
1	1	0	1	1	1	56	1
1	1	1	0	0	0	57	3
1	1	1	0	0	1	58	2
1	1	1	0	1	0	59	2
1	1	1	0	1	1	60	1
1	1	1	1	0	0	61	2
1	1	1	1	0	1	62	1
1	1	1	1	1	0	63	1
1	1	1	1	1	1	64	0

Tabela F.7: Tabela de Combinações para 6 Variáveis (parte 2).

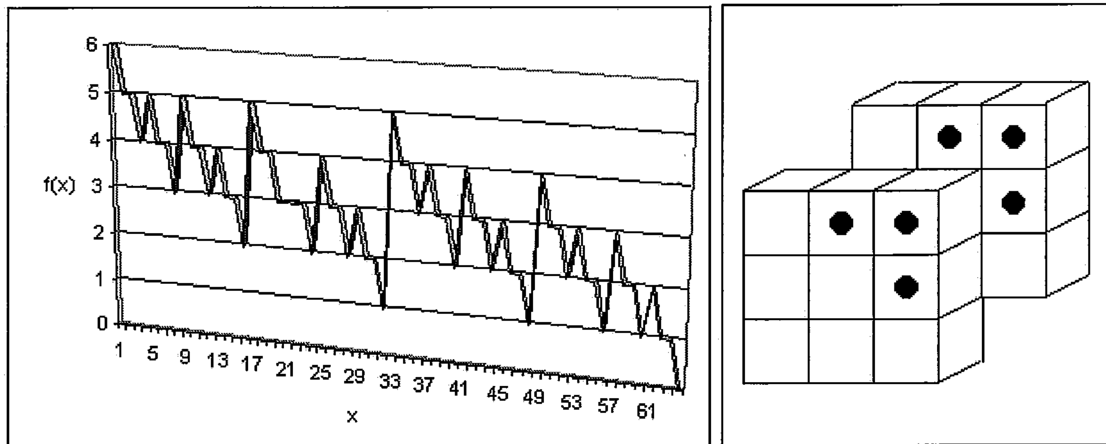


Figura 6.0.13: Espaço de Busca da Função de Energia (TESTE 68).

$x(4) \rightarrow$ pos[1][2][2]
$x(5) \rightarrow$ pos[1][3][2]
$x(6) \rightarrow$ pos[2][3][2]

As Tabelas F.6 e F.7 mostram todas as combinações das variáveis $x(1)$, $x(2)$, $x(3)$, $x(4)$, $x(5)$ e $x(6)$. Na Figura 6.0.13 foi graficado a o espaço de busca da nossa função de energia gerada e do lado se mostra o gráfico da configuração da matriz *pos* no mínimo global.

TESTE69

A função de energia gerada foi: $1 * x(1) + 1 * x(2) - 1 * x(3) * x(1) - 1 * x(4) * x(2)$. Onde:

$x(1) \rightarrow$ inn[1]
$x(2) \rightarrow$ inn[2]
$x(3) \rightarrow$ cb[1]
$x(4) \rightarrow$ cb[2]

A Tabela F.8 mostra todas as combinações das variáveis $x(1)$, $x(2)$, $x(3)$ e $x(4)$. Na Figura 6.0.13 foi representado o espaço de busca da nossa função de energia gerada e ao lado se mostra o gráfico de uma das configurações da matriz

$x1$	$x2$	$x3$	$x4$	x	$f(x)$
0	0	0	0	1	0
0	0	0	1	2	0
0	0	1	0	3	0
0	0	1	1	4	0
0	1	0	0	5	1
0	1	0	1	6	0
0	1	1	0	7	1
0	1	1	1	8	0
1	0	0	0	9	1
1	0	0	1	10	1
1	0	1	0	11	0
1	0	1	1	12	0
1	1	0	0	13	2
1	1	0	1	14	1
1	1	1	0	15	1
1	1	1	1	16	0

Tabela F.8: Tabela das Combinações para o TESTE 69.

pos no mínimo global.

TESTES: 70 e 71

A função de energia gerada foi: $1 * x(1) + 1 * x(2) + 1 * x(3) + 1 * x(4) - 2 * x(3) * x(1) - 2 * x(4) * x(2)$.

Onde:

$x(1) \rightarrow$ inn[1]

$x(2) \rightarrow$ inn[2]

$x(3) \rightarrow$ cb[1]

$x(4) \rightarrow$ cb[2]

A Tabela F.9 mostra todas as combinações das variáveis $x(1)$, $x(2)$, $x(3)$ e $x(4)$. Na Figura 6.0.13 foi graficado a o espaço de busca da nossa função de energia gerada e do lado se mostra o gráfico de uma das configurações da matriz *pos* no mínimo global.

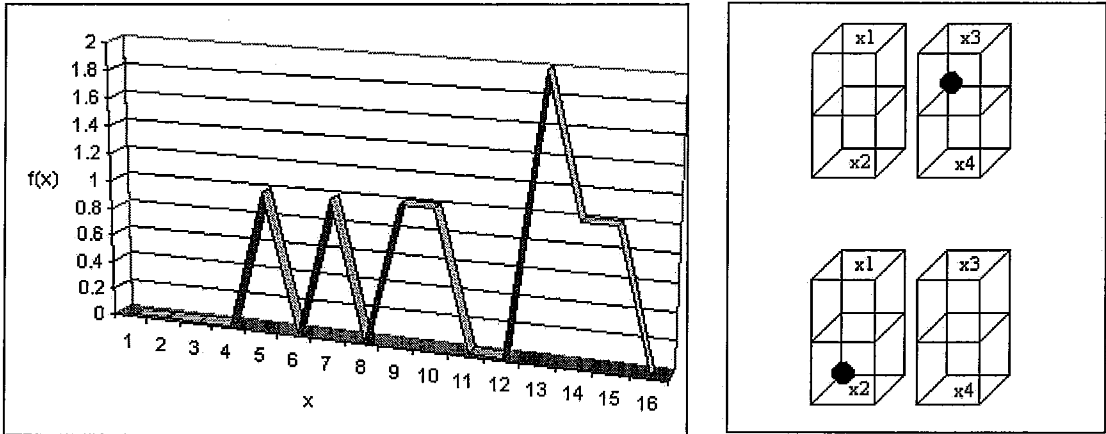


Figura 6.0.14: Espaço de Busca da Função de Energia (TESTE 69).

$x1$	$x2$	$x3$	$x4$	x	$f(x)$
0	0	0	0	1	0
0	0	0	1	2	1
0	0	1	0	3	1
0	0	1	1	4	2
0	1	0	0	5	1
0	1	0	1	6	0
0	1	1	0	7	2
0	1	1	1	8	1
1	0	0	0	9	1
1	0	0	1	10	2
1	0	1	0	11	0
1	0	1	1	12	1
1	1	0	0	13	2
1	1	0	1	14	1
1	1	1	0	15	1
1	1	1	1	16	0

Tabela F.9: Tabela das Combinações para o TESTE 70.

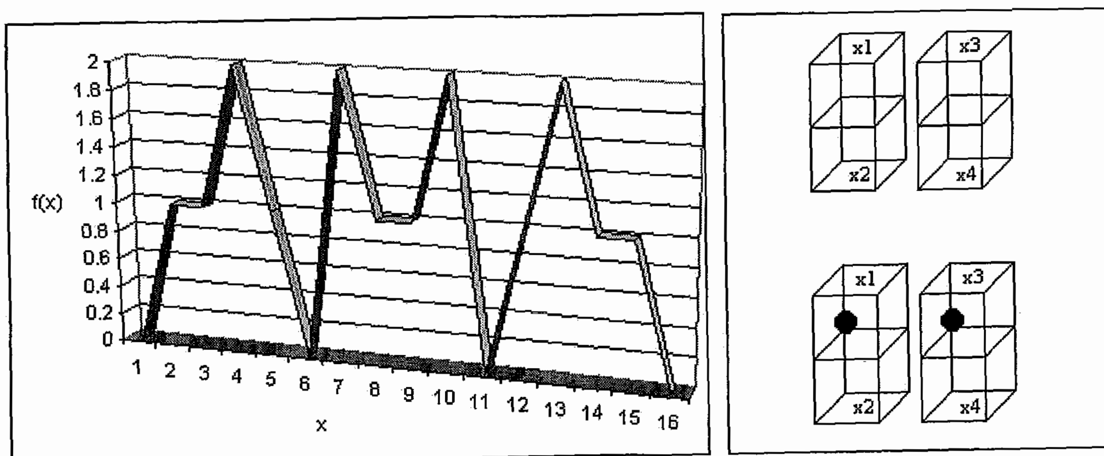


Figura 6.0.15: Espaço de Busca da Função de Energia (TESTE 70 e 71).