



COPPE/UFRJ

SATYRUS2: COMPILANDO ESPECIFICAÇÕES DE RACIOCÍNIO LÓGICO

Bruno França Monteiro

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Priscila Machado Vieira Lima

Rio de Janeiro
Abril de 2010

SATYRUS2: COMPILANDO ESPECIFICAÇÕES DE RACIOCÍNIO LÓGICO

Bruno França Monteiro

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) 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.

Examinada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Priscila Machado Vieira Lima, Ph.D.

Prof. Adilson Elias Xavier, D.Sc.

Prof. Wilson Rosa de Oliveira, D.Sc.

RIO DE JANEIRO, RJ – BRASIL

ABRIL DE 2010

Monteiro, Bruno França

SATyrus2: Compilando Especificações de Raciocínio Lógico/Bruno França Monteiro. – Rio de Janeiro: UFRJ/COPPE, 2010.

XI, 97 p.: il.; 29,7cm.

Orientadores: Felipe Maia Galvão França

Priscila Machado Vieira Lima

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2010.

Referências Bibliográficas: p. 77 – 79.

1. Compiladores. 2. Satisfabilidade. 3. Otimização.
4. ARQ-PROP II. I. França, Felipe Maia Galvão *et al.*
II. Universidade Federal do Rio de Janeiro, COPPE,
Programa de Engenharia de Sistemas e Computação. III.
Título.

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

SATYRUS2: COMPILANDO ESPECIFICAÇÕES DE RACIOCÍNIO LÓGICO

Bruno França Monteiro

Abril/2010

Orientadores: Felipe Maia Galvão França
Priscila Machado Vieira Lima

Programa: Engenharia de Sistemas e Computação

Apresenta-se, nesta dissertação, o SATyrus2, um compilador que se baseia no problema da Satisfabilidade booleana para traduzir modelos escritos sob a forma de restrições em problemas de programação linear inteira. SATish, uma linguagem declarativa de especificação, é fornecida como instrumento de modelagem para os problemas a serem resolvidos pelo SATyrus2. Por fim, a correção do compilador é avaliada por meio de testes realizados com a ARQ-PROP II, uma arquitetura capaz de realizar raciocínio proposicional por intermédio da aplicação do Princípio da Resolução.

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

SATYRUS2: COMPILING LOGIC REASONING SPECIFICATIONS

Bruno França Monteiro

April/2010

Advisors: Felipe Maia Galvão França

Priscila Machado Vieira Lima

Department: Systems Engineering and Computer Science

In this work we present SATyrus2, a compiler that is based on the Boolean Satisfiability problem. SATyrus2 can compile models written in the form of restrictions in binary integer programming instances. Satish, a declarative programming language, is provided as a tool for modeling problems that can be solved by SATyrus2. Finally, ARQ-PROP II, a limited-depth propositional reasoner, is used to test the correctness of SATyrus2.

Sumário

Lista de Figuras	viii
Lista de Tabelas	x
1 Introdução	1
1.1 Objetivos e Contribuições	2
1.2 Estrutura do Documento	2
2 Fundamentos	3
2.1 Satisfatibilidade e Forma Normal Conjuntiva	3
2.2 Mapeando Satisfatibilidade em Programação Inteira 0-1	5
2.2.1 Regras de Mapeamento	5
2.2.2 Restrições	7
2.2.3 Penalidades	8
2.2.4 Primeiro exemplo: mapeando o TSP	9
2.2.5 Segundo exemplo: mapeando o problema de Coloração de Grafos	11
2.3 SATyrus	13
3 Concepção do SATyrus2	17
3.1 Motivações para a criação de um novo compilador	17
3.2 SATyrus2	17
3.2.1 A linguagem SATish	18
3.3 Modelando o TSP	21
3.4 Modelando a Coloração de Grafos	26
4 Implementação do SATyrus2	30
4.1 A nova plataforma	30
4.1.1 Escolha da linguagem de programação	30
4.1.2 Modularização	31
4.1.3 Expansão da linguagem	35
4.2 Funcionamento do compilador	38
4.2.1 Análise léxica	38

4.2.2	Análise sintática	40
4.2.3	Geração da função de energia	42
4.3	Interface com softwares de otimização	43
4.3.1	AMPL	45
4.3.2	Xpress	51
5	Especificações de Raciocínio Lógico	54
5.1	ARQ-PROP II	54
5.1.1	Estruturas	55
5.1.2	Restrições	56
5.1.3	Compilação	60
5.2	Testes	60
5.3	Avaliação qualitativa	67
6	Conclusões	69
6.1	Sumário	69
6.2	Trabalhos futuros	70
	Referências Bibliográficas	77
A	SATish - Language Specification	80
B	Manual do compilador	85
B.1	Obtendo o código fonte do compilador	85
B.2	Instalação	86
B.3	Opções de compilação	86
B.4	Exemplos	87
B.5	Erros de compilação	88
C	Código fonte da ARQ-PROP II em linguagem SATish	93

Lista de Figuras

2.1	Fluxo de dados através dos módulos do SATyrus.	14
2.2	Rede de Hopfield de alta ordem correspondente à função de energia ϕ^*	15
2.3	Funcionamento do simulador do SATyrus.	15
3.1	Utilização do SATyrus2	18
3.2	Uma instância de TSP com quatro cidades.	22
3.3	Uma quinta cidade, cópia de C_1 , é acrescentada ao grafo da Figura 3.2.	22
3.4	Resultado do TSP modelado na Listagem 3.4.	25
3.5	<i>tour</i> obtido na solução da Figura 3.4.	25
3.6	Instância do problema de Coloração de Grafos.	26
3.7	Resultado do problema de Coloração de Grafos modelado na Listagem 3.5.	28
3.8	Solução obtida na Figura 3.7.	28
4.1	Fluxo de dados através dos módulos do compilador SATyrus2.	32
4.2	Exemplo de arquivo de entrada de dados no SATyrus.	36
4.3	Primeiro passo: negação da fórmula original; segundo passo: conversão para a FNC.	40
4.4	Fórmula resultante da restrição da Listagem 4.7.	41
4.5	Processo de obtenção da função de energia (P_i denota o valor da penalidade no nível de restrições i).	43
4.6	Integração entre SATyrus2, AMPL e Resolvedor.	46
4.7	Modelo AMPL gerado a partir do código da Listagem 4.9.	48
4.8	Arquivo AMPL de controle para o código da Listagem 4.9.	48
4.9	Resultado da execução do modelo da Figura 4.7.	50
4.10	Modelo Mosel gerado a partir do código da Listagem 4.9.	52
4.11	Resultado da execução do modelo da Figura 4.10.	53
5.1	Resumo do modelo AMPL para o código da ARQ-PROP II.	61
5.2	Resumo do modelo Mosel (Xpress) para a ARQ-PROP II.	62
5.3	Resultado esperado para a base de cláusulas ϕ'	64
5.4	Resultado obtido pelo SATyrus2 para a base de cláusulas ϕ'	65

6.1	Dados de entrada para a modelagem da Listagem 6.1.	71
6.2	Exemplo de arquivo de controle AMPL que resolve a Listagem 6.1. .	72
6.3	Solução encontrada por AMPL para o modelo da Listagem 6.1. . . .	73
B.1	Lista de opções disponíveis no SATyrus2	86

Lista de Tabelas

2.1	Tabela verdade da Fórmula 2.1	4
2.2	Exemplos de <i>tour</i>	9
4.1	Exemplos de geração de funções de energia.	34
4.2	Palavras reservadas da linguagem do SATyrus2	39
4.3	Símbolos presentes na linguagem do SATyrus2	39
4.4	Demais <i>tokens</i> presentes na linguagem do SATyrus2	39
4.5	Fórmulas geradas com a instanciação do índice j	41
4.6	Fórmulas geradas com a instanciação do índice i	41
5.1	Exemplo de configuração de CLCOMP.	55
5.2	Custos de ambas as soluções (esperada e obtida) para cada uma das restrições da modelagem anotada da ARQ-PROP II.	67
B.1	Exemplo de simplificação de função de energia	87

Lista de Listagens

3.1	Especificação de estruturas em SATish.	19
3.2	Exemplo de restrição de integridade em SATish.	20
3.3	Especificação de penalidades na linguagem SATish.	21
3.4	Instância do TSP modelada em linguagem SATish.	23
3.5	Instância do problema de Coloração de Grafos modelada em SATish.	26
4.1	Exemplo de utilização da função <i>eval</i> : o código acima imprime o inteiro 2 quando executado.	31
4.2	Exemplo de inicialização de estruturas no SATyrus.	35
4.3	Nova sintaxe para inicialização de estruturas.	36
4.4	Exemplo de indexação inválida na linguagem do primeiro compilador SATyrus.	37
4.5	Solução para o problema de indexação do SATyrus.	37
4.6	Restrição de integridade inválida para o SATyrus.	38
4.7	Exemplo de restrição integridade modelada em linguagem SATish.	40
4.8	Exemplo de restrição de integridade multiplicada por uma constante.	42
4.9	TSP de duas cidades modelado em SATish.	44
6.1	TSP modelado em linguagem AMPL.	70
6.2	Sugestão de sintaxe para o quantificador de ou-exclusivo.	76

Capítulo 1

Introdução

Problemas de otimização compõem uma grande área de pesquisas dentro da Ciência da Computação. Tais problemas consistem em achar a melhor solução, denominada solução ótima, dentre todas as soluções disponíveis.

A resolução exata desses problemas, isto é, a busca por uma solução ótima, é frequentemente difícil. Pode-se optar por métodos de aproximação que não garantem a melhor dentre todas as soluções à disposição. Alternativamente, pode-se escolher por pagar o custo computacional de uma solução ótima. Problemas de otimização trazem ainda uma dificuldade de *modelagem*: a elaboração de especificações tais que possam ser compreendidas com facilidade tanto pelo homem quanto pela máquina.

SATyrus [21] é uma plataforma neural destinada à especificação e resolução de problemas de otimização, e que combina conceitos de Redes de Hopfield de Alta Ordem, *Simulated Annealing* e Satisfatibilidade. SATyrus oferece uma linguagem de modelagem através da qual é possível expressar problemas por meio de *restrições*; estas representam as propriedades lógicas do problema a ser resolvido. SATyrus atua como um compilador, traduzindo as restrições em uma função objetivo que representa o espaço de soluções do problema original, posteriormente mapeada em uma rede neural de alta ordem. Através da aplicação do algoritmo de *Simulated Annealing*, a rede converge para o mínimo global da função objetivo, que equivale à solução ótima do problema original.

Este trabalho apresenta SATyrus2, uma nova abordagem para resolução de problemas de otimização. Ainda que aproveite a linguagem de modelagem definida durante o desenvolvimento de SATyrus, a busca por soluções ótimas em SATyrus2 não é levada a cabo por meio de redes neurais de alta ordem, e sim através de softwares de otimização como AMPL [2] e Xpress [3]. SATyrus2 se aproveita da eficácia desses softwares na busca por um mínimo global para a função objetivo.

SATyrus2 foi projetado em uma estrutura de camadas através das quais o problema original é sucessivamente traduzido em diversos formatos, desde sua formulação em lógica proposicional, passando pela função objetivo correspondente e termi-

nando em um modelo computacional escrito nas linguagens de modelagem AMPL [5] ou Mosel [4] (linguagem utilizada pela plataforma Xpress).

As seções seguintes descrevem os objetivos, as contribuições e a estrutura desta tese.

1.1 Objetivos e Contribuições

Os objetivos deste trabalho são a concepção e o desenvolvimento de uma plataforma de resolução de problemas de otimização denominada SATyrus2. A plataforma é composta por um compilador capaz de traduzir especificações lógicas em uma função objetivo a ser minimizada por meio de softwares de otimização pré-existentes. Também será concebida uma linguagem declarativa de modelagem, denominada SATish, que permite a especificação de problemas de otimização por meio de restrições que representem as propriedades lógicas do problema original. Deseja-se, por fim, compilar e executar a modelagem da ARQ-PROP II, uma arquitetura neural que implementa raciocínio lógico proposicional por meio do Princípio da Resolução.

1.2 Estrutura do Documento

Este documento se divide em seis capítulos. O Capítulo 2 apresenta a fundamentação teórica utilizada neste trabalho. Neste Capítulo são discutidos os conceitos de Satisfatibilidade, Satisfatibilidade Máxima e Minimização de Energia, que deram origem ao compilador SATyrus2. O Capítulo 3 traz a concepção do novo compilador e os motivos pelos quais optou-se por reconstruir a plataforma de modelagem concebida por SATyrus. Ainda no Capítulo 3, é apresentada a linguagem SATish, utilizada ao longo desse trabalho como ferramenta de modelagem para problemas de otimização.

O Capítulo 4 se aprofunda na plataforma SATyrus2, trazendo detalhes sobre o processo de desenvolvimento do compilador e de suas interfaces com os softwares de otimização AMPL e XPRESS. Os resultados da compilação e execução da ARQ-PROP II se encontram no Capítulo 5.

Finalmente, o Capítulo 6 traz conclusões a respeito dos rumos e resultados desta tese, assim como sugestões de trabalhos que podem nascer a partir da pesquisa realizada aqui.

Capítulo 2

Fundamentos

Este capítulo apresenta os fundamentos teóricos sobre os quais se baseia este trabalho. São discutidos os conceitos de Satisfatibilidade e Forma Normal Conjuntiva, Problemas Pseudo-Booleanos e a integração entre os mesmos e o problema da Satisfatibilidade. Por fim, um método de modelagem de problemas de otimização por intermédio de restrições lógicas é apresentado e exemplificado na forma de dois dos mais comuns problemas de otimização: o TSP (*Travelling Salesperson Problem*) e o problema da Coloração de Grafos.

2.1 Satisfatibilidade e Forma Normal Conjuntiva

O problema da Satisfatibilidade Booleana (geralmente referido como SAT) é um problema de decisão que consiste em encontrar uma atribuição de valores às variáveis proposicionais pertencentes a uma fórmula booleana qualquer de modo a torná-la verdadeira (satisfeita). Caso não exista tal atribuição, a fórmula é dita *insatisfatível*.

Fórmulas bem formadas (ou simplesmente, *fórmulas*) são expressões que satisfazem as seguintes leis de formação:

- uma variável proposicional é uma fórmula;
 - se p é uma variável proposicional, sua negação, $\neg p$, também é uma fórmula;
 - se p e q são fórmulas e \circ é um operador binário, então $(p \circ q)$ é uma fórmula.
- Os operadores binários são: $\{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow\}$.

Dada uma fórmula qualquer, o problema da Satisfatibilidade pode ser enunciado da seguinte maneira: é possível atribuir valores booleanos ($\{V, F\}$) às variáveis proposicionais de tal maneira que a fórmula seja avaliada como verdadeira? Por exemplo, considere a Fórmula 2.1:

$$p \vee (q \wedge \neg r) \tag{2.1}$$

A Tabela 2.1 representa a *tabela verdade* da Fórmula 2.1. Através dela, pode-se constatar que existem cinco atribuições de valores que tornam a Fórmula 2.1 verdadeira. No contexto do problema da Satisfatibilidade, cada uma dessas cinco atribuições é denominada uma *solução* para o problema.

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

Tabela 2.1: Tabela verdade da Fórmula 2.1

O problema da Satisfatibilidade Booleana é NP-Completo [15]. Um caso especial para esse problema é a situação na qual a fórmula a ser satisfeita se encontra na Forma Normal Conjuntiva (FNC). Uma fórmula está na Forma Normal Conjuntiva quando é composta por conjunções de *cláusulas*. Cláusulas são disjunções de *literais*; estes, por sua vez, são variáveis proposicionais ou negações de variáveis proposicionais. p e $\neg r$ são exemplos de literais. A disjunção destes literais, $(p \vee \neg q)$, é um exemplo de cláusula.

Uma cláusula é *satisfeita* caso exista ao menos uma atribuição de valores às variáveis pertencentes à cláusula que a torna verdadeira. Considere a Fórmula 2.2, composta por seis cláusulas:

$$p \wedge \neg p \wedge (\neg p \vee q) \wedge \neg q \wedge (\neg p \vee r) \wedge \neg r \quad (2.2)$$

A fórmula é insatisfatível, ou seja, não existe uma atribuição de valores às variáveis p , q e r que torne a fórmula verdadeira como um todo. No entanto, a atribuição $p = F$, $q = V$ e $r = V$ satisfaz a segunda, a terceira e a quinta cláusulas presentes em 2.2. Pode-se formular o seguinte problema de otimização: dada uma fórmula escrita na Forma Normal Conjuntiva, qual é a atribuição de valores booleanos às suas variáveis que maximiza o número de cláusulas satisfeitas? Este problema, denominado Problema da Satisfatibilidade Máxima (MAX-SAT), tem implicações em diversas áreas de pesquisa, tais como os problemas de checagem de modelos, *field-programmable gate arrays* (FPGAs), roteamento em redes etc.

O problema de MAX-SAT funcionará como base para o desenvolvimento teórico das seções a seguir.

2.2 Mapeando Satisfatibilidade em Programação Inteira 0-1

Um problema de Programação 0-1 (ou problema Pseudo-Booleano) é um problema de programação linear inteira no qual todas as variáveis estão restritas aos valores 0 e 1. Problemas de Programação 0-1 são compostos por uma função objetivo a ser otimizada e por um conjunto de variáveis binárias, da seguinte maneira:

calcule:

$$\min_{x \in X} f(x) \quad \left(\text{ou } \max_{x \in X} f(x) \right)$$

onde:

$$f : \{0, 1\}^n \rightarrow \mathbb{R}$$

$$X \subseteq \{0, 1\}^n.$$

A natureza booleana destes problemas os conecta diretamente ao problema de Satisfatibilidade. De fato, é possível converter problemas de MAX-SAT em problemas Pseudo-Booleanos por meio de um mapeamento entre fórmulas lógicas e valores contidos no conjunto $\{0, 1\}$. As regras que constituem esse mapeamento são apresentadas a seguir.

2.2.1 Regras de Mapeamento

Considere o seguinte mapeamento, inicialmente proposto por Gadi Pinkas [20] e posteriormente aperfeiçoado por Lima *et al* [19], que relaciona valores booleanos aos inteiros 0 e 1:

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

As regras acima relacionam expressões lógicas escritas na Forma Normal Conjuntiva a valores inteiros pertencentes ao intervalo $[0, 1]$. Mais especificamente, o *verdadeiro* booleano é mapeado em 1, enquanto proposições *falsas* são mapeadas em 0. No contexto do problema de programação pseudo-booleana que pode ser obtido por meio desse mapeamento, apenas duas soluções são possíveis: 1, caso a fórmula original seja satisfeita; e 0, em caso contrário. Isso implica na necessidade

de *maximizar* a função objetivo construída pelo mapeamento, uma vez que, dentre os dois possíveis valores de uma solução, 1 é o valor mais alto. Caso se queria *minimizar* a função objetivo, o mapeamento deve ser aplicado à *negação* da fórmula booleana original. Desse modo, dada uma fórmula ϕ escrita na FNC, pode-se obter um problema de programação pseudo-booleana caso se aplique o mapeamento descrito acima à fórmula $\neg\phi$: o problema consistirá em minimizar a função objetivo construída através da conversão entre fórmulas e expressões algébricas. Caso uma solução com custo igual a 0 seja encontrada, a fórmula original ϕ é satisfeita.

O mapeamento apresentado aqui relaciona SAT ao problema de Programação Linear 0-1 por meio da construção de uma função objetivo - daqui em diante designada **função de energia** - obtida através da aplicação da função H à negação da fórmula que se deseja satisfazer, ou seja:

$$E = H(\neg\phi)$$

Considere o seguinte exemplo trivial: deseja-se satisfazer a Fórmula 2.3. Para tal, é necessário encontrar ao menos uma atribuição de valores booleanos às variáveis proposicionais a e b de forma a tornar a Fórmula 2.3 verdadeira ($\phi = V$).

$$\phi = \neg a \vee b \tag{2.3}$$

A função de energia que corresponde à Fórmula 2.3 pode ser obtida por meio da aplicação direta das regras de mapeamento definidas na página 5. A aplicação destas regras, em conjunto com um breve desenvolvimento lógico das proposições obtidas, dá origem à seguinte função de energia:

$$\begin{aligned} E &= H(\neg\phi) \\ &= H(\neg(\neg a \vee b)) \\ &= H(a \wedge \neg b) \\ &= H(a) \times H(\neg b) \\ &= \underbrace{H(a) \times (1 - H(b))}_{\text{função de energia}} \end{aligned}$$

Substituindo (por motivos de clareza) os símbolos $H(a)$ e $H(b)$ por a^* e b^* na função de energia obtida, o problema original, isto é, satisfazer a Fórmula 2.3, fica convertido no seguinte problema de programação linear pseudo-booleana:

minimize:

$$E = a^* \times (1 - b^*)$$

onde:

$$a^* \in \{0, 1\}$$

$$b^* \in \{0, 1\}$$

Resolver o problema acima é trivial: $a^* = 0$ é uma das possíveis soluções. Mas:

$$\begin{aligned} a^* = 0 & \implies (\text{por substituição de símbolos}) \\ H(a) = 0 & \implies (\text{pela segunda regra de mapeamento}) \\ a = F \end{aligned}$$

É fácil observar que $a = F$ é, de fato, uma solução viável para o problema de satisfatibilidade da Fórmula 2.3.

O mapeamento descrito na página 5 oferece apenas dois tipos de solução: 0, caso uma solução exata para o problema (isto é, a satisfação de toda a fórmula) for encontrada; e 1, caso não exista tal solução. É possível alterar essa estratégia de modo a adaptá-la ao problema de MAX-SAT, modificando-a de forma a aumentar o número de soluções possíveis e, assim, inserir uma medida de *grau de não-satisfatibilidade* da solução obtida: o quão perto determinada solução está da solução exata.

Seja φ uma fórmula escrita na Forma Normal Conjuntiva, ou seja, $\varphi = \bigwedge_i \varphi_i$, onde $\varphi_i = \bigvee_j p_{ij}$ e p_{ij} é um literal. Portanto, $\neg\varphi = \bigvee \gamma_i$, onde $\gamma_i = \neg\varphi_i$. Introduzimos a função H^* e substituímos a equação $E = H(\neg\varphi)$ por $E = H^*(\neg\varphi) = \sum_i H(\gamma_i)$. Portanto, $E = \sum_i H(\bigwedge_j \neg p_{ij}) = \sum_i \prod_j H(\neg p_{ij})$. Dessa forma, E conta o número de cláusulas que não são satisfeitas por uma solução qualquer, ou seja, a função H^* é uma ferramenta de conversão de problemas de SAT em problemas de MAX-SAT, que podem ser posteriormente convertidos em problemas de programação booleana.

A função H^* funcionará como base para a estratégia de resolução de problemas descrita neste trabalho. Entretanto, antes que aplicações práticas de H^* possam ser apresentadas, é necessário falar sobre restrições e penalidades.

2.2.2 Restrições

A aplicação do mapeamento apresentado na página 5 requer uma especificação clausal do problema a ser resolvido. É necessário que o problema original seja convertido em uma instância de SAT, associando-se variáveis pertencentes ao problema a variáveis proposicionais que formam expressões lógicas escritas na FNC. Tais expressões são denominadas **restrições**, e constituem a base das modelagens de problemas expostas neste trabalho.

As restrições são representações clausais das propriedades do problema a ser resolvido. Elas se dividem em dois tipos:

- **restrições de integridade:** descrevem o formato de uma solução *viável* para um dado problema. As restrições de integridade devem ser negadas antes da

aplicação do mapeamento descrito na página 5, e fazem com que o valor da função de energia diminua à medida em que são satisfeitas.

- **restrições de otimalidade:** atribuem custos (pesos) às soluções viáveis de um dado problema, e são utilizadas na obtenção de uma solução *ótima* para o problema original. As restrições de otimalidade não são negadas antes da aplicação do mapeamento descrito na página 5, e fazem com que o valor da função de energia aumente à medida em que são satisfeitas.

O formato das restrições será esclarecido na Seção 2.2.4, que apresenta um exemplo de modelagem do Problema do Caixeiro Viajante (*Travelling Salesperson Problem*).

2.2.3 Penalidades

Penalidades são constantes multiplicativas que organizam as restrições por ordem de prioridade dentro da função de energia. Restrições com penalidades mais altas possuem chances menores de serem insatisfeitas durante o processo de resolução de uma função de energia qualquer.

A atribuição de penalidades se dá por meio da divisão do conjunto de restrições de um dado problema em diferentes *níveis*: cada nível possui uma penalidade distinta associada a ele. Pode-se pensar a respeito das penalidades como se fossem pesos: elas definem o quão importante é satisfazer um determinado conjunto de cláusulas em detrimento a outras (que possuem penalidade mais baixa) dentro de um problema de MAX-SAT.

Seja n_i o número de cláusulas presentes nas restrições de nível i , e seja ϵ um valor pequeno. As penalidades (v_i) associadas aos diferentes níveis de restrições são calculadas da seguinte forma:

$$\begin{aligned} v_0 &= 1 \\ v_1 &= ((n_0 + 1) \times v_0) + \epsilon, \text{ onde } \epsilon \text{ é um valor pequeno} \\ v_2 &= ((n_0 + 1) \times v_0) + ((n_1 + 1) \times v_1) + \epsilon \\ &\vdots \\ v_k &= ((n_0 + 1) \times v_0) + \dots + ((n_{k-1} + 1) \times v_{k-1}) + \epsilon \end{aligned}$$

As penalidades são calculadas de modo a tornar mais vantajosa a satisfação de cláusulas que pertençam a níveis de penalidade mais elevados. Mais especificamente, para um nível i qualquer, seja v_i o valor numérico da penalidade associada a i . O valor da penalidade associada ao nível $i + 1$ é igual a $((n_i + 1) \times v_i) + \epsilon$. Desse modo, violar todas as n_i cláusulas presentes em i custa $(n_i \times v_i)$, ao passo que violar apenas uma cláusula pertencente ao nível $i + 1$ custa o equivalente a $((n_i + 1) \times v_i) + \epsilon$.

De posse do mapeamento apresentado na Seção 2.2.1 e do conceito de penalidades associadas aos diferentes níveis de restrições, é possível modelar problemas de otimização utilizando a estratégia descrita nas seções anteriores. Dentre esses problemas está o Problema do Caixeiro Viajante (*Travelling Salesperson Problem*), abordado na próxima seção.

2.2.4 Primeiro exemplo: mapeando o TSP

O Problema do Caixeiro Viajante (TSP) pode ser enunciado da seguinte maneira: dado um conjunto de m cidades e os valores das distâncias entre quaisquer pares de cidades, deseja-se saber qual é o caminho de menor custo que percorra todas as cidades exatamente uma vez.

Para que se possa modelar o problema, é necessário converter a especificação do TSP descrita em linguagem natural em um conjunto de restrições sobre o qual se possa aplicar o mapeamento descrito na Seção 2.2.1. Começamos a modelar o TSP através de um grafo não-direcionado $G = (V, A)$, onde V é um conjunto de vértices de G e A é o seu conjunto de arestas. Associando cada vértice i a uma cidade pertencente ao problema, as arestas $(i, j) \in A$ passam a representar as conexões entre as diferentes cidades do problema original.

Para que o *tour* obtido como solução comece e termine na mesma cidade, a cidade inicial é replicada e todos os custos correspondentes às suas arestas de incidência são replicados também. A distância entre a cidade inicial e sua cópia é 0, o que força o retorno à origem. O problema original passa a ter, dessa maneira, $n = m + 1$ cidades.

Seja M um conjunto de n^2 variáveis proposicionais v_{ij} nas quais i representa uma cidade (vértice de V) e j a posição dessa cidade no *tour* solução, e seja $dist_{ij} \in \mathbb{Z}$ o custo associado à aresta $(i, j) \in A$. Uma solução viável para o problema é um subconjunto de M no qual todas as variáveis proposicionais possuam valor verdadeiro (V), e que represente uma sequência de cidades a serem percorridas no *tour* resultante. A Tabela 2.2 apresenta exemplos de atribuições de valores que resultam em percursos nos quais as cidades são visitadas nas ordens $C_1C_2C_3$ e $C_2C_3C_1$. Resolver o TSP consiste em obter uma sequência semelhante que represente o *tour* de custo mais baixo.

Sequência de cidades	Atribuição de valores
$C_1C_2C_3$	$v_{11} = v_{22} = v_{33} = V$
$C_2C_3C_1$	$v_{21} = v_{32} = v_{13} = V$

Tabela 2.2: Exemplos de *tour*.

As restrições que modelam o problema do TSP, propostas em [27], são as seguin-

tes:

Restrições de integridade:

- (i) Todas as n cidades devem fazer parte do *tour*:

$$\forall i \exists j | 1 \leq i \leq n, 1 \leq j \leq n : v_{ij}. \text{ Então, } \varphi_1 = \bigwedge_i (\bigvee_j (v_{ij})).$$

- (ii) Duas cidades não podem ocupar a mesma posição no *tour*:

$$\forall i \forall j \forall i' | 1 \leq i \leq n, 1 \leq j \leq n, 1 \leq i' \leq n, i \neq i' : \neg(v_{ij} \wedge v_{i'j}).$$

Então, $\varphi_2 = \bigwedge_i \bigwedge_{i \neq i'} \bigwedge_j \neg(v_{ij} \wedge v_{i'j}).$

- (iii) Uma cidade não pode ocupar mais de uma posição no *tour*:

$$\forall i \forall j \forall j' | 1 \leq i \leq n, 1 \leq j \leq n, 1 \leq j' \leq n, j \neq j' : \neg(v_{ij} \wedge v_{ij'}).$$

Então, $\varphi_3 = \bigwedge_i \bigwedge_j \bigwedge_{j \neq j'} \neg(v_{ij} \wedge v_{ij'}).$

Restrições de otimalidade:

- (iv) O custo entre duas cidades consecutivas no *tour*:

$$\forall i \forall j \forall i' | 1 \leq i \leq n, 1 \leq j \leq n-1, 1 \leq i' \leq n, i \neq i' : dist_{ii'}(v_{ij} \wedge v_{i'(j+1)}).$$

Então, $\varphi_4 = \bigvee_i \bigvee_{i \neq i'} \bigvee_{j \leq n-1} dist_{ii'}(v_{ij} \wedge v_{i'(j+1)}).$

A cada restrição listada acima está associado um peso (penalidade) que corresponde a uma constante multiplicativa na equação de energia. É importante observar que, para a restrição (i), deseja-se que exista uma e *apenas uma* posição na qual uma certa cidade possa estar ao longo do *tour* ótimo. Isto é, deseja-se que apenas uma das disjunções seja verdadeira para cada um dos possíveis valores de i , o que constitui um *ou-exclusivo*. De modo geral, restrições da forma $\exists i | 1 \leq i \leq n : F_i$ são representações compactas da disjunção $F_1 \vee F_2 \vee \dots \vee F_n$. Caso se deseje que apenas uma das fórmulas F_i seja verdadeira, pode-se definir restrições do tipo:

$$\forall i, j | 1 \leq i \leq n, 1 \leq j \leq n, i \neq j : F_i \rightarrow \neg F_j$$

Restrições como a apresentada acima são denominadas restrições WTA (*Winner Takes All*). As restrições WTA têm o papel de implementar a operação de *ou-exclusivo*, ausente no mapeamento definido por Pinkas. Na modelagem do TSP apresentada anteriormente, as restrições (ii) e (iii) são WTAs. Estas são associadas ao grau de penalidade mais alto, o que previne a quebra da exclusividade desejada para as disjunções presentes na restrição (i).

As restrições de otimalidade têm peso mais baixo, de valor numérico igual a 1. Sejam α e β os valores da penalidades associadas à restrição (i) e às restrições WTA, respectivamente. O cálculo de α e β é feito como a seguir:

$$\begin{cases} dist &= \max\{dist_{ij}\} \\ \alpha &= ((n^3 - 2n^2 + n + 1) * dist) + \epsilon \\ \beta &= ((n^2 + 1) * \alpha) + \epsilon \end{cases}$$

Pode-se agora aplicar as regras de mapeamento apresentadas na Seção 2.2.1:

$$E_i = \alpha H^*(\neg\varphi_1) + \beta H^*(\neg\varphi_2) + \beta H^*(\neg\varphi_3)$$

Como $\varphi_1 = \bigwedge_i (\bigvee_j (v_{ij}))$, $\neg\varphi_1 = \bigvee_i (\bigwedge_j (\neg v_{ij}))$. Logo:

$$H^*(\neg\varphi_1) = \sum_{i=1}^n H(\bigwedge_j (\neg v_{ij})) = \sum_{i=1}^n \prod_{j=1}^n H(\neg v_{ij}) = \sum_{i=1}^n \prod_{j=1}^n (1 - v_{ij}).$$

No entanto, devido às restrições WTA, o mapeamento de $\neg\varphi_1$ pode ser simplificado em:

$$H_{WTA}^*(\neg\varphi_1) = \sum_{i=1}^n \sum_{j=1}^n (1 - v_{ij}).$$

Como $\neg\varphi_2 = \bigwedge_i \bigwedge_{i' \neq i} \bigwedge_j \neg(v_{ij} \wedge v_{i'j})$:

$$H^*(\neg\varphi_2) = \sum_{i=1}^n \sum_{i'=1, i' \neq i}^n \sum_{j=1}^n H(\neg(v_{ij} \wedge v_{i'j})) = \sum_{i=1}^n \sum_{i'=1, i' \neq i}^n \sum_{j=1}^n v_{ij} v_{i'j}$$

Como $\neg\varphi_3 = \bigwedge_i \bigwedge_j \bigwedge_{j' \neq j} \neg(v_{ij} \wedge v_{ij'})$:

$$H^*(\neg\varphi_3) = \sum_{i=1}^n \sum_{j=1}^n \sum_{j'=1, j' \neq j}^n H(\neg(v_{ij} \wedge v_{ij'})) = \sum_{i=1}^n \sum_{j=1}^n \sum_{j'=1, j' \neq j}^n v_{ij} v_{ij'}$$

O próximo passo é calcular a parcela da função de energia que corresponde às restrições de otimalidade:

$$\begin{aligned} E_o &= \sum_s H^*(\varphi_4) \\ H^*(\varphi_4) &= \sum_{i=1}^n \sum_{i'=1, i' \neq i}^n \sum_{j=1}^{n-1} dist_{ii'} H(v_{ij} \vee v_{i'(j+1)}) = \\ &= \sum_{i=1}^n \sum_{i'=1, i' \neq i}^n \sum_{j=1}^{n-1} dist_{ii'} v_{ij} v_{i'(j+1)}. \end{aligned}$$

A equação de energia completa para o problema do TSP corresponde a $E = E_i + E_o$.

2.2.5 Segundo exemplo: mapeando o problema de Coloração de Grafos

O problema da Coloração de Grafos consiste em encontrar a menor atribuição de cores aos vértices de um grafo qualquer de modo que dois vértices vizinhos não estejam associados à mesma cor.

Seja $G = (V, A)$ um grafo não-direcionado de n vértices. Os conjuntos de variáveis proposicionais utilizados no mapeamento do problema de Coloração de Grafos são: V_{colour} , de tamanho n^2 , tal que cada um dos seus elementos vc_{ik} é verdadeiro caso a cor k esteja atribuída ao vértice i ; C_{colour} , de tamanho n , para o qual c_k é verdadeiro se a cor k estiver atribuída a um vértice qualquer; e $neigh$, que contém n^2 elementos, utilizado para indicar relações de vizinhança entre dois vértices quaisquer: $neigh_{ii'}$

é verdadeiro caso o vértice i seja vizinho do vértice i' .

As restrições que definem o problema da Coloração são as seguintes:

Restrições de integridade:

(i) Todas os vértices devem ter estar associados a alguma cor:

$$\forall i \exists k | 1 \leq i \leq n, 1 \leq k \leq n: vc_{ik}. \text{ Então, } \varphi_1 = \bigwedge_i (\bigvee_k (vc_{ik})).$$

(ii) Dois vértices vizinhos não podem ter a mesma cor:

$$\forall i \forall i' \forall k | 1 \leq i \leq n, 1 \leq i' \leq n, 1 \leq k \leq n, i \neq i': \neg neigh_{ii'} \vee \neg (vc_{ik} \wedge vc_{i'k}).$$

$$\text{Logo, } \varphi_2 = \bigwedge_i \bigwedge_{i' \neq i} \bigwedge_k (\neg neigh_{ii'} \vee \neg (vc_{ik} \wedge vc_{i'k})).$$

(iii) Um vértice não pode estar associado a mais de uma cor:

$$\forall i \forall k \forall k' | 1 \leq i \leq n, 1 \leq k \leq n, 1 \leq k' \leq n, k \neq k': \neg (vc_{ik} \wedge vc_{ik'}).$$

$$\text{Logo, } \varphi_3 = \bigwedge_i \bigwedge_k \bigwedge_{k' \neq k} \neg (vc_{ik} \wedge vc_{ik'}).$$

(iv) Caso uma cor k esteja associada a um vértice i qualquer, então c_k deve ser verdadeiro:

$$\forall i \forall k | 1 \leq i \leq n, 1 \leq k \leq n: \neg vc_{ik} \vee c_k.$$

$$\text{Logo, } \varphi_4 = \bigwedge_i \bigwedge_k \neg vc_{ik} \vee c_k.$$

Restrições de otimalidade:

(v) O número de cores utilizadas deve ser mínimo:

$$\forall k | 1 \leq k \leq n: c_k. \text{ Então, } \varphi_5 = \bigvee_k c_k.$$

No mapeamento acima, (iii) é uma WTA que garante a validade do ou-exclusivo para as disjunções presentes em (i). A restrição (iii) é associada ao valor de penalidade mais alto, β , enquanto as demais restrições de integridade são associadas à penalidade α . A restrição de otimalidade (v) possui o peso mais baixo, de valor igual 1.

Em um procedimento similar ao caso do TSP, as penalidades são calculadas como a seguir:

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

A função de energia resultante é uma composição das parcelas referentes às restrições de integridade (E_i) e otimalidade (E_o). De acordo com as restrições listadas anteriormente, $E_i = \alpha(H_{WTA}^*(\neg\varphi_1) + H^*(\neg\varphi_2) + H^*(\neg\varphi_4)) + \beta(H^*(\neg\varphi_3))$ e $E_o = \sum_s H^*(\varphi_5)$.

Portanto:

$$\begin{aligned}
E_i &= \beta \left(\sum_{i=1}^n \sum_{k=1}^n \sum_{k'=1, k' \neq k}^n v c_{ik} v c_{ik'} \right) + \alpha \left(\sum_{i=1}^n \sum_{k=1}^n (1 - v c_{ik}) \right) + \alpha \left(\sum_{i=1}^n \sum_{i'=1, i' \neq i}^n \sum_{k=1}^n v c_{ik} v c_{i'k} \text{neigh}_{ii'} \right) + \\
&\quad \alpha \left(\sum_{i=1}^n \sum_{k=1}^n v c_{ik} (1 - c_k) \right) \\
E_o &= \sum_{k=1}^n c_k
\end{aligned}$$

A função de energia resultante é expressa por $E = E_i + E_o$.

2.3 SATyrus

A plataforma SATyrus, concebida por LIMA *et al.* [21], é uma plataforma neural destinada à resolução de problemas de otimização. SATyrus combina conceitos de Redes de Hopfield de Alta Ordem, *Simulated Annealing* e Satisfatibilidade com o intuito de fornecer um ambiente de modelagem e resolução de problemas.

SATyrus atua como um compilador: o código de entrada é uma especificação lógica de um problema qualquer, escrita sob a forma de restrições semelhantes às apresentas na seção anterior, a partir das quais se pode aplicar o mapeamento descrito na página 5. O código objeto é a função de energia corresponde à especificação fornecida como entrada para o compilador.

SATyrus também atua como um otimizador: o código objeto resultante do estágio de compilação é convertido em uma rede neural de alta ordem que funciona como um resolvidor para equação de energia construída a partir do problema original.

SATyrus fornece uma linguagem de programação declarativa com a qual é possível especificar problemas de otimização sob a forma de restrições. Uma vez modelado na linguagem fornecida por SATyrus, o problema alvo passa por um processo de compilação. Nesse processo, o compilador de SATyrus converte o conjunto de restrições em uma função de energia equivalente. A função de energia é então convertida em uma rede estocástica de Hopfield de alta ordem; esta atua como um otimizador que visa encontrar o mínimo global para o problema original [6].

Neste trabalho, *SATish*, a linguagem de modelagem provida por SATyrus, foi expandida e aperfeiçoada, assunto que será discutido no Capítulo 4. A especificação completa da linguagem SATish se encontra no Apêndice A.

A plataforma SATyrus se divide em dois módulos:

- **compilador:** responsável por aplicar o mapeamento descrito na Seção 2.2.1 às restrições especificadas no arquivo de entrada fornecido pelo usuário. A aplicação do mapeamento dá origem a uma função de energia a ser minimizada;

- **simulador:** encarregado de criar e resolver a rede neural correspondente à função de energia obtida pelo compilador. O simulador usa o algoritmo de *Simulated Annealing* para percorrer o espaço de soluções em busca de um mínimo global.

A Figura 2.1 ilustra o fluxo de dados através dos dois módulos que compõem o SATyrus.

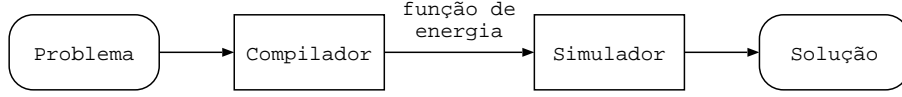


Figura 2.1: Fluxo de dados através dos módulos do SATyrus.

O simulador de SATyrus, desenvolvido por PEREIRA [22], mapeia a função de energia obtida pelo compilador em uma rede de Hopfield de alta ordem. A rede é composta por um conjunto de neurônios que correspondem, um a um, às variáveis presentes na função de energia. Para todos os estados da rede, neurônios ligados equivalem a variáveis proposicionais de valor verdadeiro na especificação lógica do problema; neurônios desligados correspondem a variáveis cujo valor booleano é falso. As cláusulas presentes especificação lógica do problema, que são traduzidas em multiplicações entre variáveis na função de energia, são mapeadas em ligações entre dois ou mais neurônios da rede. O peso de cada uma dessas ligações é igual à constante multiplicativa que acompanha as multiplicações entre variáveis. Por exemplo, seja:

$$\phi^* = 3pq - 3pr - 8qpr - 3qr$$

uma função de energia hipotética obtida pelo compilador de SATyrus. A rede neural de alta ordem que equivale a ϕ^* está ilustrada na Figura 2.2, onde N_x simboliza o neurônio criado por uma variável x qualquer.

Como as cláusulas existentes na especificação do problema podem conter três ou mais variáveis proposicionais cada uma, as ligações equivalentes podem relacionar três ou mais neurônios simultaneamente: estas são denominadas *ligações de alta ordem*. A presença de tais ligações não constitui um problema, visto que redes de alta ordem também convergem para um ótimo global [23].

A Figura 2.3 ilustra o funcionamento do simulador de SATyrus.

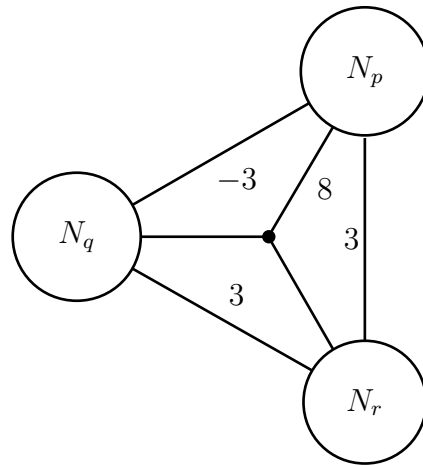


Figura 2.2: Rede de Hopfield de alta ordem correspondente à função de energia ϕ^* .

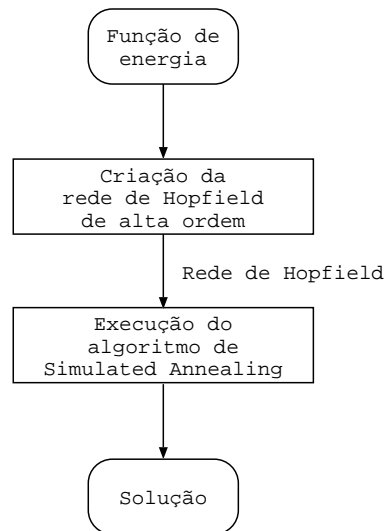


Figura 2.3: Funcionamento do simulador do SATyrus.

A plataforma SATyrus, ainda que funcional, apresentava limitações que motivaram a criação de um novo compilador. Um exemplo de tais limitações é a utilização de um único método - as redes de alta ordem - na minimização da função de energia. Outras limitações estão descritas no próximo capítulo, que traz a apresentação do SATyrus2, a nova plataforma de resolução de problemas de otimização apresentada neste trabalho.

Capítulo 3

Concepção do SATyrus2

Este capítulo apresenta a plataforma SATyrus2: seu processo de concepção, as motivações que levaram à criação de um novo compilador e a linguagem SATish, projetada para funcionar como formato padrão dos modelos a serem utilizados em conjunto com o SATyrus2. A última seção ilustra o funcionamento do compilador através de dois modelos exemplo escritos em linguagem SATish.

3.1 Motivações para a criação de um novo compilador

A plataforma SATyrus, discutida na Seção 2.3, apresentava limitações que restringiam e dificultavam seu uso. Por exemplo, certas características da linguagem de modelagem reconhecida por SATyrus demandavam a escrita de código desnecessário por parte do usuário, assunto que será abordado no Capítulo 4. Além disso, a baixa modularização do código fonte da plataforma SATyrus também se apresentava como uma dificuldade: o único módulo de simulação impossibilitava a utilização de estratégias diversas de resolução para a equação de energia.

Com o intuito de superar essas limitações, um novo compilador foi criado.

3.2 SATyrus2

SATyrus2 foi projetado com o objetivo principal de aproveitar a eficácia de diversos softwares de resolução de problemas matemáticos disponíveis, tais como AMPL e Xpress. Ao contrário de SATyrus, a plataforma SATyrus2 não atua como um resolvedor: softwares externos se encarregam da tarefa de minimizar as equações de energia provenientes dos modelos fornecidos pelo usuário.

SATyrus2 é uma completa reimplementação do compilador SATyrus. Ainda que projetados para resolver os mesmos tipos de problemas, os dois compiladores

são inteiramente distintos: o código fonte original de SATyrus não foi aproveitado durante o desenvolvimento da nova plataforma.

SATyrus2 funciona da seguinte maneira: inicialmente, o usuário deve optar por um dos softwares de otimização suportados pela nova plataforma. O problema alvo deve ser modelado em linguagem SATish, e então compilado no SATyrus2. O processo de compilação consiste em traduzir a função de energia construída a partir do modelo de entrada para a linguagem de modelagem do software de otimização escolhido pelo usuário. Essa tradução resulta em um segundo modelo, que serve como entrada para o software de otimização escolhido; este será responsável por minimizar a equação de energia e obter uma solução viável para o problema. A Figura 3.1 ilustra esse processo:

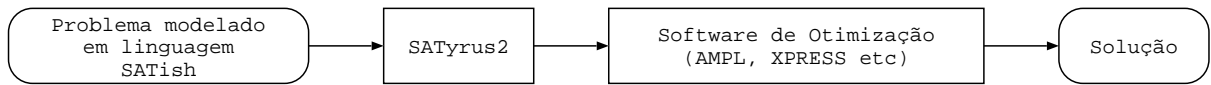


Figura 3.1: Utilização do SATyrus2

Os detalhes a respeito da implementação do SATyrus2 serão discutidos no Capítulo 4. Também será discutida no Capítulo 4 a integração entre o SATyrus2 e os softwares de otimização AMPL e XPRESS.

3.2.1 A linguagem SATish

A proposta de implementação de um novo compilador exigia a criação (ou adoção) de uma linguagem de programação que permitisse expressar de modo automatizado as estratégias de modelagem descritas na Seção 2.2.1. A linguagem SATish surgiu a partir de uma expansão da sintaxe e dos princípios estabelecidos pela linguagem de modelagem construída durante o desenvolvimento de SATyrus. SATish suporta a criação de constantes e estruturas de valores, bem como a especificação de restrições e a atribuição de penalidades a cada uma delas.

SATish é linguagem de programação declarativa: a lógica da computação é expressa sem que um fluxo de controle seja estabelecido pelo usuário. Modelos escritos em SATish descrevem apenas as propriedades lógicas do problema a ser resolvido. Uma vez dentro do compilador, os modelos são avaliados e convertidos nas funções de energia que correspondam às suas variáveis, restrições e penalidades.

Os modelos escritos em SATish são divididos em três seções: **(i)** definição de constantes e estruturas; **(ii)** definição de restrições; e **(iii)** atribuição de penalidades. Estas seções são discutidas nos próximos parágrafos.

Definição de constantes e estruturas

Em SATish, *constantes* são identificadores que simbolizam números inteiros. As constantes funcionam como nomes atribuídos pelo usuário a determinados valores; por exemplo, é possível atribuir o nome x ao valor 1, e dessa forma utilizar o símbolo x como substituto para o inteiro 1 ao longo do código fonte. Em SATish, as constantes podem ser utilizadas em operações aritméticas, alocações de estruturas e iterações sobre intervalos.

As *estruturas* são conjuntos indexados de valores (constantes ou variáveis), utilizadas pelo usuário para agrupar valores correlacionados. As estruturas fornecidas pela linguagem SATish funcionam como tabelas *hash*: elas consistem em associações entre índices e valores. Em SATish, os índices de estruturas são listas de inteiros que representam uma determinada posição em uma matriz multidimensional. Por exemplo, o índice ‘1, 4, 2’ se refere ao elemento m_{142} de uma estrutura M qualquer. Quanto aos valores, estes podem ser de dois tipos: constantes numéricas e variáveis proposicionais.

Assim como as constantes, as estruturas também são ligadas a identificadores que funcionam como referências para as mesmas ao longo do código fonte. A Listagem 3.1 apresenta um exemplo de definição de constantes e estruturas em linguagem SATish:

```
1   n = 2;
2
3   M(n, n+1);
4   M = { 1, 1: 1; 1, 2: 0; 1, 3: 1;
5         2, 1: 1; 2, 2: 0; 2, 3: 1 };

```

Listagem 3.1: Especificação de estruturas em SATish.

A primeira linha da Listagem 3.1 contém uma definição de constante: o símbolo n é vinculado ao inteiro 2. Na linha 3, a constante n é utilizada na alocação da estrutura M , definida com 2 linhas e 3 colunas. As duas últimas linhas da Listagem 3.1 apresentam a inicialização da estrutura M : nesta estrutura, $m_{11} = 1, m_{12} = 0, m_{13} = 1$ etc.

A linguagem SATish não impõe limitações às dimensões das estruturas. Estas ficam restritas, portanto, ao espaço de armazenamento à disposição do usuário.

Restrições

As restrições são representações compactas de propriedades lógicas e aritméticas de valores presentes no código fonte. As restrições disponíveis na linguagem SATish

são implementações das restrições apresentadas na Seção 2.2.2, transcritas em uma linguagem de modelagem declarativa.

Assim como as constantes e as estruturas, as restrições também são atreladas a identificadores: estes funcionam como referências para os grupos de restrições utilizados durante o cálculo de penalidades. As restrições podem ser de dois tipos: restrições de integridade e restrições de otimalidade. As restrições de integridade são antecedidas pela palavra chave **intgroup**; as restrições de otimalidade, por **optgroup**.

A Listagem 3.2 apresenta uma típica especificação de restrição de integridade em linguagem SATish:

```

1  intgroup int0 :
2      forall{i , j} where i in (1 , n) , j in (1 , n) and i != j :
3      M(i , j) and N(i , j)  $\rightarrow$  P(i , j) ;

```

Listagem 3.2: Exemplo de restrição de integridade em SATish.

A primeira linha da Listagem 3.2 define uma restrição de integridade que pertence ao grupo de restrições denominado *int0*. As linhas 2 e 3 constituem o corpo da restrição: a linha 2 define um par de *índices* e um par de *intervalos* sobre os quais os índices iteram. Em SATish, todos os intervalos são fechados e definidos como uma tupla de constantes ou inteiros que indicam os limites do intervalo. Ainda é possível especificar uma *condição* a ser satisfeita durante o processo de iteração: no exemplo da Listagem 3.2, a iteração ocorrerá apenas para valores de *i* e *j* diferentes.

A linha 3 contém uma fórmula lógica que relaciona três estruturas (*M*, *N* e *P*) e seus respectivos elementos. Durante o processo de geração da função de energia, os índices são eliminados da fórmula que compõe a restrição, em num processo denominado *instanciação de índices*, descrito no Capítulo 4. A instanciação dá origem a relações diretas entre os elementos das estruturas presentes na restrição. No exemplo da Listagem 3.2, seja $n = 3$. Como *i* e *j* iteram ambos sobre o intervalo $[0, n]$ e $i \neq j$, a restrição da Listagem 3.2 dá origem às seguintes cláusulas:

$$m_{12} \wedge n_{12} \Rightarrow p_{12}$$

$$m_{13} \wedge n_{13} \Rightarrow p_{13}$$

$$m_{21} \wedge n_{21} \Rightarrow p_{21}$$

$$m_{23} \wedge n_{23} \Rightarrow p_{23}$$

$$m_{31} \wedge n_{31} \Rightarrow p_{31}$$

$$m_{32} \wedge n_{32} \Rightarrow p_{32}$$

Cada restrição, seja ela uma restrição de integridade ou otimalidade, deve ser vinculada a um determinado *grupo*, que será utilizado durante o cálculo de penalidades. O não cumprimento dessa exigência é um erro sintático que será reportado ao usuário.

Penalidades

As penalidades são constantes multiplicativas que são aplicadas às expressões geradas pelo mapeamento descrito na Seção 2.2.1. Em SATish, as penalidades são atribuídas a *grupos de restrições*; estes, por sua vez, são definidos ao longo do código fonte, à medida em que restrições de integridade e otimalidade são declaradas.

A Listagem 3.3 apresenta um exemplo de definição de penalidades:

```

1  penalties :
2      int0 level 0;
3      int1 level 1;
4      wtA  level 2;

```

Listagem 3.3: Especificação de penalidades na linguagem SATish.

A palavra chave **penalties** abre a seção de definição de penalidades. Nas linhas subsequentes, os grupos de penalidade são, um a um, relacionados a um valor inteiro antecedido pela palavra chave **level**. Estes valores ordenam os grupos em níveis de penalidades: quanto maior o valor, maior é o nível de penalidade ao qual o grupo de restrições é relacionado. Assim, no exemplo da Listagem 3.3, o grupo *wtA* se encontra em um nível de penalidade superior ao nível do grupo *int1*, que por sua vez está em um nível de penalidade mais alto que *int0*, uma vez que $2 > 1 > 0$.

De posse de uma linguagem de modelagem (SATish) e de um compilador capaz de compilá-la (SATyrus2), é possível construir modelos computacionais equivalentes a problemas de otimização reais, tomando como base as estratégias apresentadas no Capítulo 2. As Seções 3.3 e 3.4 trazem exemplos de possíveis modelagens.

3.3 Modelando o TSP

A modelagem do TSP consiste em traduzir para a linguagem SATish a representação lógica do problema apresentada na Seção 2.2.4. É necessário, em primeiro lugar, criar estruturas que representem os agrupamentos de variáveis proposicionais presentes naquela modelagem. São utilizadas duas estruturas: **pos**, que armazena as posições das diversas cidades ao longo do *tour*; e *dist*, que contém os valores das distâncias entre todos os pares de cidades que fazem parte do problema.

Definidas as estruturas, as restrições da página 2.2.4 devem ser traduzidas para a linguagem SATish. No modelo, as restrições devem ser reunidas em diferentes grupos de penalidades; estes ao final do código fonte, serão relacionados, um a um, a um nível de prioridades distinto.

Considere a instância do TSP ilustrada na Figura 3.2:

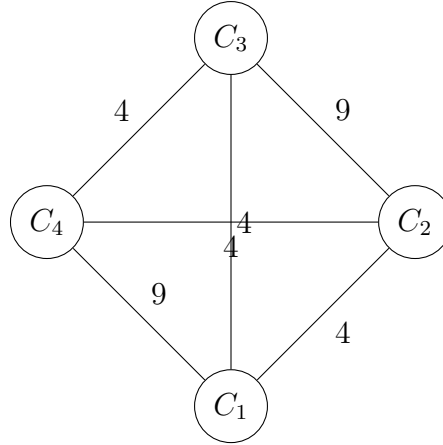


Figura 3.2: Uma instância de TSP com quatro cidades.

Queremos calcular o *tour* ótimo que passe por todas as quatro cidades representadas na ilustração: C_1 , C_2 , C_3 e C_4 . Para que o *tour* comece e termine na mesma cidade, criamos uma cidade auxiliar, C_5 , réplica da cidade C_1 . As distâncias entre C_5 e as demais cidades são copiadas uma a uma a partir das distâncias entre C_1 e as cidades restantes do problema. O grafo que representa a disposição final das cidades para esta instância do TSP está ilustrado na Figura 3.3:

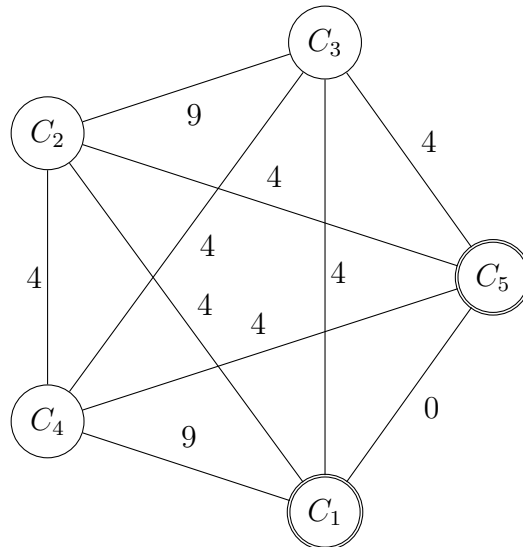


Figura 3.3: Uma quinta cidade, cópia de C_1 , é acrescentada ao grafo da Figura 3.2.

De posse de todas essas informações, o TSP pode ser modelado em linguagem SATish de acordo com código transcrito na Listagem 3.4:

```

n=5;
pos(n,n);
dist(n,n);

pos = [
    1,1: 1;
    5,5: 1
];

dist = [
    1,1: 0; 1,2: 4; 1,3: 4; 1,4: 9; 1,5: 0;
    2,1: 4; 2,2: 0; 2,3: 9; 2,4: 4; 2,5: 4;
    3,1: 4; 3,2: 9; 3,3: 0; 3,4: 4; 3,5: 4;
    4,1: 9; 4,2: 4; 4,3: 4; 4,4: 0; 4,5: 9;
    5,1: 0; 5,2: 4; 5,3: 4; 5,4: 9; 5,5: 0
];

intgroup int0:
    forall{i,j} where i in (1,n), j in (1,n):
        pos[i][j];

intgroup wta:
    forall{i,j,k}
        where i in (1,n), j in (1,n), k in (1,n) and i != k:
            pos[i][j] -> not pos[k][j];

intgroup wta:
    forall{i,j,l}
        where i in (1,n), j in (1,n), l in (1,n) and j != l:
            pos[i][j] -> not pos[i][l];

optgroup cost:
    forall{i,j,k}
        where i in (1,n), j in (2,n), k in (1,n) and i != k:
            dist[i][k] (pos[i][j] and pos[k][j-1]);

optgroup cost:
    forall{i,j,k}

```

```

where  $i$  in  $(1, n)$ ,  $j$  in  $(1, n-1)$ ,  $k$  in  $(1, n)$  and  $i \neq k$ :
     $\text{dist}[i][k]$  ( $\text{pos}[i][j]$  and  $\text{pos}[k][j+1]$ );

```

penalties :

```

cost level 0;
int0 level 1;
wta level 2;

```

Listagem 3.4: Instância do TSP modelada em linguagem SATish.

No modelo apresentado na Listagem 3.4, a estrutura **dist** armazena os pesos das arestas presentes no grafo da Figura 3.3. Os pesos representam as distâncias entre as diversas cidades que compõem o problema, e serão utilizados durante o processo de busca por uma solução ótima. A definição da estrutura **pos** traz as cidades C_1 e C_5 fixadas, respectivamente, como pontos de partida e de chegada do *tour* a ser construído. Os demais elementos de **pos** representam as posições das outras cidades ao longo do caminho, e funcionam como variáveis do problema a ser resolvido.

O resultado da compilação do modelo definido na Listagem 3.4 é um segundo modelo, escrito na linguagem de entrada do software de otimização escolhido pelo usuário do SATyrus2. Todo o processo de execução do segundo modelo em um software de otimização é descrito no Capítulo 4. Por hora, é suficiente analisar o resultado final obtido a partir do modelo descrito na Listagem 3.4. A solução está apresentada na Figura 3.4.

A solução obtida consiste em uma atribuição de valores 0 e 1 às variáveis proposicionais pertencentes à estrutura **pos**. De acordo com o mapeamento estabelecido na Seção 2.2.1, o inteiro 1 corresponde ao *verdadeiro* booleano, enquanto o inteiro 0 representa as proposições *falsas*. Portanto, se $\text{pos}_{ij} = 1$ para um par i e j qualquer, então é verdade que a cidade i está na j -ésima posição do *tour* encontrado.

Na solução da Figura 3.4, as variáveis que receberam o valor 1 durante o processo de minimização da função de energia são: p_{22} , p_{43} e p_{34} . Sabe-se também que as variáveis p_{11} e p_{55} têm valor igual a 1, pois foram fixadas na inicialização da estrutura **pos**. Dessa maneira, o *tour* obtido pelo SATyrus2 corresponde à ilustração da Figura 3.5, e constitui, de fato, uma solução ótima para o problema.

```

:  _varname[i] _var[i]   :=
1   pos_1_2      0
2   pos_1_3      0
3   pos_1_4      0
4   pos_1_5      0
5   pos_5_4      0
6   pos_5_1      0
7   pos_5_2      0
8   pos_5_3      0
9   pos_3_4      1
10  pos_3_5      0
11  pos_3_2      0
12  pos_3_3      0
13  pos_3_1      0
14  pos_4_5      0
15  pos_4_4      0
16  pos_4_1      0
17  pos_4_3      1
18  pos_4_2      0
19  pos_2_5      0
20  pos_2_4      0
21  pos_2_3      0
22  pos_2_2      1
23  pos_2_1      0
;

```

Figura 3.4: Resultado do TSP modelado na Listagem 3.4.

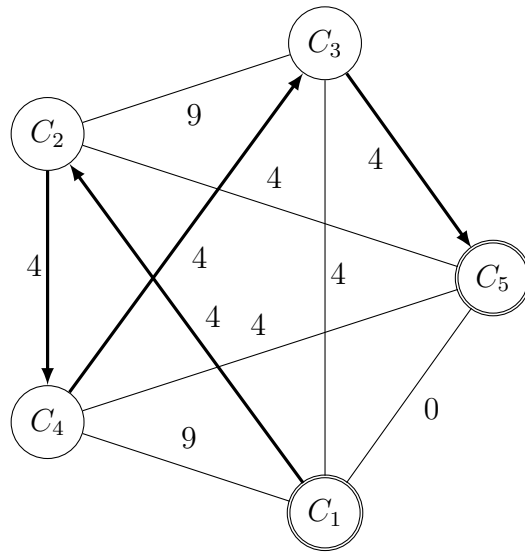


Figura 3.5: *tour* obtido na solução da Figura 3.4.

3.4 Modelando a Coloração de Grafos

A tradução para a linguagem SATish do problema da Coloração de Grafos, modelado na Seção 2.2.5, demanda a criação de três estruturas de variáveis: **vc**, **col** e **neigh** são equivalentes, respectivamente, aos conjuntos V_{colour} , C_{colour} e $neigh$, presentes na modelagem do Capítulo 2. Para um grafo de n vértices, **vc** tem $n_x n$ elementos. Se $vc(i, j) = 1$, então é verdade que ao vértice i foi atribuída a cor denominada j . De forma semelhante, $col(j) = 1$ apenas se a cor j for utilizada na resolução do problema.

A título de exemplo, considere o grafo representado na Figura 3.6:

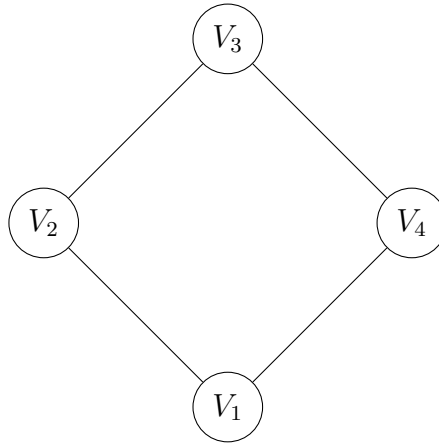


Figura 3.6: Instância do problema de Coloração de Grafos.

O código da Listagem 3.5 implementa a modelagem desta instância do problema da Coloração. Os dados de entrada são as adjacências do grafo da Figura 3.6, representadas pelos elementos da estrutura *neigh*. Na listagem abaixo, $neigh(i, j) = 1$ caso o vértice i seja vizinho do vértice j .

```
n = 4;

neigh(n, n);
vc(n, n);
col(n);

neigh = [
    1,1: 0; 1,2: 1; 1,3: 0; 1,4: 1;
    2,1: 1; 2,2: 0; 2,3: 1; 2,4: 0;
    3,1: 0; 3,2: 1; 3,3: 0; 3,4: 1;
    4,1: 1; 4,2: 0; 4,3: 1; 4,4: 0
];
```

```

intgroup int0:
  forall{i} where i in (1,n); exists{k} where k in (1,n):
    vc[i][k];

intgroup int1:
  forall{i,j,k}
    where i in (1,n), j in (1,n), k in (1,n) and i!=j:
      neigh[i][j] -> not (vc[i][k] and vc[j][k]);

intgroup int1:
  forall{i,k,l}
    where i in (1,n), k in (1,n), l in (1,n) and k!=l:
      not (vc[i][k] and vc[i][l]);

intgroup int0:
  forall{i,k} where i in (1,n), k in (1,n):
    vc[i][k] -> col[k];

optgroup cost:
  forall{k} where k in (1,n):
    col[k];

penalties:
  cost level 0;
  int0 level 1;
  int1 level 2;

```

Listagem 3.5: Instância do problema de Coloração de Grafos modelada em SATish.

A solução obtida pelo SATyrus2 para o modelo da Listagem 3.5 está ilustrada na Figura 3.7. Assim como na modelagem do TSP, a solução consiste em uma atribuição de valores 0 e 1 às variáveis presentes na modelo. A solução representada na Figura 3.7 usa apenas duas cores (col_1 e col_4), o suficiente para colorir o grafo bipartido ilustrado na Figura 3.6. A representação gráfica da solução obtida para esta instância do problema se encontra na Figura 3.8 (as cores 1 e 4 são representadas por branco e cinza, respectivamente).

```

:  _varname[i] _var[i]  :=
1   vc_4_3      0
2   vc_3_1      0
3   vc_4_1      1
4   vc_3_3      0
5   vc_3_4      1
6   vc_4_4      0
7   col_2       0
8   col_3       0
9   col_1       1
10  col_4       1
11  vc_1_4      1
12  vc_1_2      0
13  vc_1_3      0
14  vc_1_1      0
15  vc_2_1      1
16  vc_2_3      0
17  vc_2_2      0
18  vc_2_4      0
19  vc_4_2      0
20  vc_3_2      0
;

```

Figura 3.7: Resultado do problema de Coloração de Grafos modelado na Listagem 3.5.

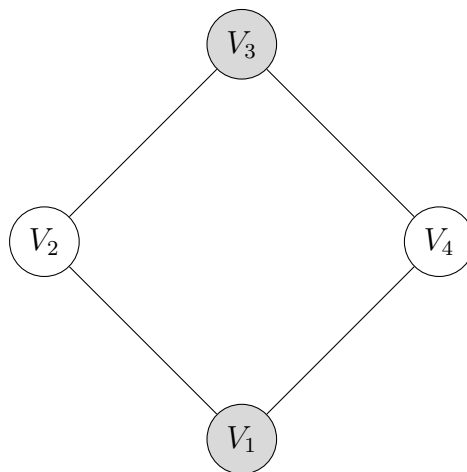


Figura 3.8: Solução obtida na Figura 3.7.

A soluções apresentadas neste capítulo foram obtidas através da integração entre o SATyrus2 e softwares de otimização pré-existentes. Tal integração, assim como o processo de implementação do SATyrus2, são discutidos no Capítulo 4.

Capítulo 4

Implementação do SATyrus2

Este capítulo aborda as características e o processo de desenvolvimento do SATyrus2: os detalhes de implementação do projeto e aspectos de funcionamento do novo compilador. Também é discutida a integração do SATyrus2 com as plataformas de otimização AMPL e Xpress.

4.1 A nova plataforma

Com a intenção de contornar os problemas de modularização da primeira implementação do SATyrus e desenvolver uma plataforma mais flexível, o SATyrus2 foi projetado de forma que todos os seus módulos operem de maneira independente. Cada módulo é responsável por implementar uma única tarefa do processo de compilação e pode ser substituído, caso seja necessário, por uma implementação alternativa da mesma tarefa.

Os detalhes do processo de desenvolvimento do código e a linguagem de programação escolhida são discutidos nas seções a seguir.

4.1.1 Escolha da linguagem de programação

Python [1] foi a linguagem de programação escolhida para o desenvolvimento do SATyrus2. Python é uma linguagem de alto-nível, dinâmica e interpretada, utilizada para propósitos gerais. Python oferece suporte aos três principais paradigmas de programação (funcional, imperativo e orientado ao objeto), ainda que seja primordialmente orientada ao objeto.

Durante o desenvolvimento do código do compilador, notou-se que a principal vantagem de Python sobre C++, linguagem utilizada no projeto do primeiro compilador SATyrus, é a presença de uma função de avaliação dinâmica de código. Em Python, essa função chama-se *eval* [12]. *eval* é uma função que avalia *strings* de maneira semelhante à forma como expressões são avaliadas dentro do contexto de

uma linguagem de programação. O código da Listagem 4.1, escrito em linguagem Python, exemplifica o uso dessa função:

```
print(eval("1 + 1"))
```

Listagem 4.1: Exemplo de utilização da função *eval*: o código acima imprime o inteiro 2 quando executado.

No código da Listagem 4.1, graças à função **eval**, a *string* “1 + 1” é avaliada e interpretada por Python de forma similar à avaliação e interpretação de expressões quaisquer que compõem a gramática da linguagem. Desse modo, a expressão “eval(“1 + 1”)” é equivalente à expressão “1 + 1”, por sua vez equivalente ao inteiro 2. O resultado da execução do código representado na Listagem 4.1 é, portanto, a impressão do inteiro 2.

A presença da função *eval* trouxe ao desenvolvimento do SATyrus2 uma grande facilidade: trechos de código escritos em SATish poderiam ser armazenados em memória no formato de *strings* e diretamente entregues ao interpretador Python para execução através da função *eval*. Este recurso permite que trechos de código escritos em linguagem SATish sejam convertidos em código Python e possam usufruir de todos os recursos presentes na linguagem interpretada (algoritmos embutidos na linguagem, checagem de erros etc). C++ não possui recurso equivalente.

Uma provável perda de desempenho proveniente da escolha de Python em detrimento a C++ não foi levada em consideração, uma vez que, como será demonstrado no Capítulo 5, o tempo de compilação de modelos escritos em SATish não é significativo frente ao tempo necessário para minimizar a função de energia gerada a partir de tais modelos, trabalho realizado por softwares externos ao compilador SATyrus2.

Escolhida a linguagem de programação, o SATyrus2 foi concebido em uma arquitetura de módulos, cada qual responsável por uma tarefa específica do processo de compilação. A modularização da plataforma é discutida na próxima seção.

4.1.2 Modularização

O código fonte do SATyrus2 é dividido em quatro módulos: Interface com o usuário, Parser, Gerador da função de energia e Interface com resolvedores. O fluxo de dados através dos módulos é ilustrado na Figura 4.1.

Os parágrafos seguintes trazem breves descrições a respeito do funcionamento dos quatro módulos que compõem o compilador. A Seção 4.2, por sua vez, apresenta uma análise mais aprofundada a respeito da modularização do SATyrus2 e da implementação de seus módulos.

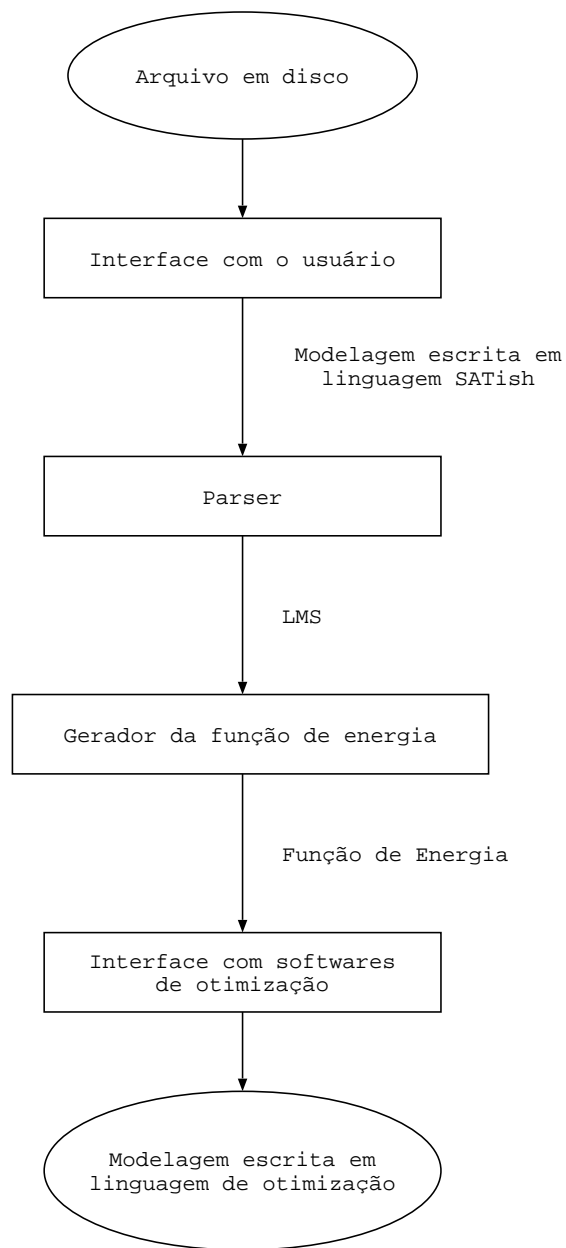


Figura 4.1: Fluxo de dados através dos módulos do compilador SATyrus2.

Interface com o usuário

Responsável pela interação do SATyrus2 com o ambiente externo, o módulo de Interface com o usuário provê opções de compilação e instruções a respeito da utilização das mesmas. Este módulo também é o responsável por ler a partir do disco o arquivo de modelagem especificado pelo usuário como entrada para o compilador.

Erros nas opções de compilação fornecidas pelo usuário são imediatamente reportados. Inexistência do arquivo de modelagem e escolha de opções de compilação incompatíveis são exemplos de erros desta natureza. Erros que porventura aconteçam durante o processo de compilação são disparados em forma de exceções pelos demais módulos e capturados pelo módulo de Interface com o usuário, que se encarrega de reportar os erros ocorridos. O Apêndice B traz um detalhamento dos possíveis erros de compilação retornados pelo SATyrus2.

Caso a interação com o usuário transcorra sem erros, o módulo de interface entrega ao módulo seguinte, denominado Parser, o conteúdo do arquivo de modelagem fornecido pelo usuário do compilador.

Parser

O Parser é o módulo encarregado por verificar e interpretar o código fonte dos modelos fornecidos pelo usuário. Inicialmente, o conteúdo do arquivo de modelagem é lido a partir do disco pelo módulo de Interface com o usuário, que repassa ao módulo Parser uma sequência de caracteres que corresponde exatamente ao conteúdo do arquivo lido. O Parser verifica se cada subsequência de caracteres presentes no conteúdo original do arquivo corresponde a uma sequência válida de símbolos no contexto da linguagem reconhecida pelo compilador. Este processo é denominado *análise léxica* (mais detalhes a respeito de análise léxica se encontram na Seção 4.2.1). Cada subsequência válida de caracteres é denominada um *token*.

Para que a modelagem fornecida pelo usuário seja considerada sintaticamente correta, os *tokens* devem estar dispostos em grupos considerados válidos dentro do contexto da linguagem SATish ¹. É tarefa do módulo Parser verificar a validade de tais grupos, num processo denominado *análise sintática* (mais detalhes a respeito da análise sintática realizada pelo SATyrus2 se encontram na Seção 4.2.2).

Erros na análise sintática disparam exceções que são posteriormente capturadas pelo módulo de Interface com o usuário, responsável por formatar e reportar o erros ocorridos ao usuário do compilador. Caso não haja erros sintáticos, o módulo Parser se encarrega da construção da fórmula lógica - daqui em diante, denominada *LMS* - correspondente ao conjunto de restrições presentes no código fonte fornecido pelo usuário, de acordo com as regras de mapeamento especificadas na Seção 2.2.1. Por

¹A especificação da linguagem SATish se encontra no Apêndice A

fim, a LMS é entregue ao módulo de geração da função de energia.

Gerador da função de energia

O Gerador da função de energia é o módulo responsável pela conversão da fórmula lógica construída pelo Parser em uma função de energia a ser otimizada, de acordo com as regras de mapeamento estabelecidas na Seção 2.2.1.

A função de energia é composta por um conjunto de variáveis binárias e por uma expressão algébrica que relaciona essas variáveis. As variáveis da função de energia derivam das variáveis proposicionais contidas na LMS construída durante a análise sintática, de acordo com a seguinte regra: cada variável proposicional dá origem a uma única variável na função de energia. Por outro lado, a expressão algébrica que compõe a função de energia é obtida através de um processo de substituição dos operadores lógicos por subexpressões algébricas apropriadas de acordo com o mapeamento apresentado na Seção 2.2.1. A Tabela 4.1 ilustra o processo de geração da função de energia por meio de exemplos de conversões dessa natureza.

Fórmula	Variáveis binárias	Expressão algébrica correspondente
$(a \vee b) \wedge \neg c$	a, b, c	$(a + b) * (1 - c)$
$(\neg a \vee \neg b) \wedge (a \vee c)$	a, b, c	$((1 - a) + (1 - b)) * (a + c)$
$\neg a \wedge (\neg a \vee b)$	a, b	$(1 - a) * ((1 - a) + b)$

Tabela 4.1: Exemplos de geração de funções de energia.

Uma vez concluída a geração da função de energia, esta é mantida em memória no formato de *string*. Mais adiante ela será utilizada pelo módulo de Interface com softwares de otimização durante a geração de modelos AMPL e Mosel (Xpress).

Interface com softwares de otimização

O módulo de Interface com softwares de otimização é um conjunto de submódulos independentes encarregados, cada um deles, de gerar modelagens escritas em linguagens de otimização específicas. A implementação atual do SATyrus2 conta com dois destes submódulos: o primeiro deles é capaz de gerar modelos escritos em linguagem AMPL; o segundo, modelos escritos em Mosel.

Os módulos que fazem interface com softwares de otimização requerem dois parâmetros de entrada: uma lista que contenha todas as variáveis pertencentes à função de energia e o próprio corpo da função de energia como uma sequência de caracteres. De posse desses dados, um modelo de otimização é gerado e escrito em disco. O usuário poderá utilizá-lo como parâmetro de entrada para o software de otimização de sua escolha.

Para todas as linguagens de otimização suportadas pelo SATyrus2 (atualmente, AMPL e Mosel), é necessário garantir:

- que as variáveis da função de energia sejam declaradas como binárias; ou seja, que essas variáveis só possam assumir dois valores: 0 e 1;
- que uma estratégia de otimização adequada seja especificada no modelo de saída, uma vez que a função de energia gerada pelo SATyrus2 é não-linear e não-convexa;
- que a função de energia seja minimizada (e não maximizada) pelo software de otimização;
- que a solução encontrada por esse software seja exibida para o usuário.

Essas exigências são satisfeitas através da utilização de diretivas e comandos próprios das linguagem de otimização suportadas. Mais detalhes a respeito dos modelos AMPL e Mosel gerados pelo SATyrus2 se encontram nas Seções 4.3.1 e 4.3.2, respectivamente. O formato dos modelos de otimização são determinados pelo usuário por meio de opções fornecidas pelo módulo de interface. O Apêndice B apresenta detalhes sobre todas as opções de compilação disponibilizadas pelo SATyrus2.

4.1.3 Expansão da linguagem

Durante a concepção do SATyrus2, a linguagem SATish foi expandida em relação à linguagem de modelagem utilizada pela primeira versão da plataforma SATyrus. As modificações foram feitas com o objetivo de aumentar a expressividade da linguagem e facilitar a programação por parte do usuário do compilador. Essas modificações são discutidas nos parágrafos a seguir.

Entrada de dados

As estruturas de variáveis presentes na primeira versão da linguagem SATish são declaradas dentro dos arquivos de modelagem, mas seus conteúdos são inicializados por meio de chamadas de sistema adicionais que lêem arquivos externos ao modelo. Um exemplo de inicialização desse tipo se encontra na Listagem 4.2:

```
n = 2;
M(n, n);
M read from M.txt;
```

Listagem 4.2: Exemplo de inicialização de estruturas no SATyrus.

Na primeira linha da Listagem 4.2, a constante n é definida com valor igual a 2. A segunda linha traz a declaração da estrutura M : esta possui duas dimensões, cada uma com n elementos. A terceira linha indica que o conteúdo de M deve ser

lido a partir do arquivo *M.txt*, escrito em disco. Um exemplo do formato do arquivo *M.txt* se encontra na Figura 4.2:

```
1,1-1
1,2-1
2,1-0
2,2-1
```

Figura 4.2: Exemplo de arquivo de entrada de dados no SATyrus.

No exemplo acima, os valores de $M(1,1)$, $M(1,2)$, $M(2,1)$ e $M(2,2)$ são inicializados com 1, 1, 0 e 1, respectivamente.

O SATyrus2 oferece recurso semelhante: as estruturas podem ser inicializadas a partir de arquivos externos armazenados em disco. Na versão atual da linguagem SATish, a sintaxe deste recurso é ligeiramente diferente: usa-se apenas a palavra chave **from**, seguida pelo nome do arquivo a ser lido. No entanto, de forma alternativa, os valores podem ser especificados diretamente no código fonte dos modelos, como exemplificado na Listagem 4.3:

```
n = 2;
M(n,n);
M = { 1,1: 1;
      1,2: 1;
      2,1: 0;
      2,2: 1; };
```

Listagem 4.3: Nova sintaxe para inicialização de estruturas.

As duas primeiras linhas do código da Listagem 4.3 são idênticas às linhas iniciais do código da Listagem 4.2: a primeira linha define a constante n ; a segunda declara a estrutura $M_{n \times n}$. Adicionalmente, cada uma das demais linhas define uma posição na estrutura M , através da seguinte sintaxe:

$$\langle \textit{Lista de Índices} \rangle \quad : \quad \langle \textit{Inteiro} \rangle \quad ;$$

onde:

$$\begin{aligned} \langle \textit{Lista de Índices} \rangle &::= \langle \textit{Índice} \rangle \\ &| \quad \langle \textit{Índice} \rangle , \langle \textit{Lista de Índices} \rangle \end{aligned}$$

e índices são inteiros positivos.

Na nova sintaxe, o usuário pode optar, portanto, por inicializar as estruturas a partir do próprio código fonte dos modelos.

Uso de constantes na indexação de estruturas

A linguagem de especificação de modelos de SATyrus impunha uma restrição ao programador: não era permitido que constantes numéricas funcionassem como índices nas estruturas de valores. Por exemplo, o trecho de código da Listagem 4.4 é inválido dentro da linguagem do primeiro compilador.

```
intgroup int1 :  
    Pos [ 1 ] [ 1 ] or Pos [ 1 ] [ 2 ] ;
```

Listagem 4.4: Exemplo de indexação inválida na linguagem do primeiro compilador SATyrus.

No contexto do primeiro compilador SATyrus, o problema do código acima é a utilização das constantes 1 e 2 como índices na estrutura *Pos*. SATyrus proíbe esse tipo de utilização de constantes; no entanto, é possível que o usuário necessite escrever uma restrição de integridade semelhante à apresentada no código da Listagem 4.4. Esse problema pode ser contornado através da substituição das constantes por índices que variem apenas no intervalo de valores desejado. A Listagem 4.5 traz um exemplo dessa estratégia.

```
intgroup int1 :  
    forall { i , j , k } ; 1 <= i <= 1, 1 <= j <= 1, 2 <= k <= 2 :  
        Pos [ i ] [ j ] or Pos [ i ] [ k ] ;
```

Listagem 4.5: Solução para o problema de indexação do SATyrus.

SATyrus2 resolve esse problema implementando o uso de constantes como índices nas estruturas de variáveis. O trecho de código da Listagem 4.4 é um código válido para o novo compilador.

Sintaxe expandida das restrições

SATyrus restringia a sintaxe das fórmulas presentes em restrições de integridade e otimalidade à Forma Normal Conjuntiva (FNC); ou seja, para que um arquivo de modelagem pudesse ser compilado por SATyrus, era necessário que todas as fórmulas presentes no código fonte correspondessem a conjunções de cláusulas. A Listagem 4.6 ilustra um trecho de código inválido dentro da linguagem do primeiro compilador:

```
intgroup int1 :  
  forall { i }; 1<=i<=n :  
    A[ i ] -> B[ i ] and C[ i ]
```

Listagem 4.6: Restrição de integridade inválida para o SATyrus.

SATyrus2 expande a sintaxe das fórmulas de modo que tanto o operador de implicação presente no código da Listagem 4.6 quanto a especificação não-clausal de sua restrição de integridade sejam válidos dentro da nova linguagem. SATyrus2 se encarrega de converter as fórmulas escritas pelo usuário para a FNC durante o processo de geração da função de energia.

Todos os operadores lógicos aceitos pelo compilador e a ordem de precedência sintática dos mesmos estão definidos na especificação completa da linguagem SATish presente no Apêndice A.

4.2 Funcionamento do compilador

O SATyrus2 opera de acordo com o fluxo de dados ilustrado na figura da página 32. Inicialmente, o arquivo de modelagem é lido a partir do disco pelo módulo de Interface com o usuário. O conteúdo desse arquivo é entregue ao Parser, módulo encarregado da execução das análises léxica e sintática no código fonte do arquivo de entrada e da geração da formulação lógica correspondente à modelagem fornecida pelo usuário. Uma vez completa a análise sintática, a LMS resultante é submetida ao módulo de Geração da função de energia, que aplica sobre ela o mapeamento descrito na página 5. Por fim, a função de energia é utilizada na elaboração de um modelo escrito em linguagem de otimização.

Os detalhes sobre todo esse processo se encontram nas subseções a seguir.

4.2.1 Análise léxica

A análise léxica consiste no mapeamento de sequências de caracteres em *tokens*, representações de conjuntos de símbolos aceitos pela linguagem reconhecida pelo compilador. Um *token* pode ser, por exemplo, um algarismo, uma sequência de algarismos ou mesmo uma sequência de algarismos e caracteres alfabéticos que formam o nome de uma constante ou variável dentro código fonte.

O analisador léxico do SATyrus2 classifica os *tokens* em três conjuntos: palavras reservadas, símbolos e demais *tokens*. As palavras reservadas são sequências de caracteres especiais que compõem parte da gramática da linguagem reconhecida pelo compilador e não podem ser usadas como identificadores, já que possuem precedência

durante o processo de análise léxica. A Tabela 4.2 traz a lista de todas as palavras reservadas presentes na linguagem SATish.

and	exists	forall
from	in	intgroup
level	not	optgroup
or	penalties	where

Tabela 4.2: Palavras reservadas da linguagem do SATyrus2

Os símbolos são caracteres ou sequências de caracteres que, assim como as palavras reservadas, também fazem parte da gramática reconhecida pelo compilador. A Tabela 4.3 lista todos os símbolos presentes na gramática da linguagem SATish.

;	=	(
)	,	==
!=	[]
:	{	}
->	<-	<->
+	-	*
/		

Tabela 4.3: Símbolos presentes na linguagem do SATyrus2

Os demais *tokens* são definidos através de expressões regulares. A Tabela 4.4 traz uma listagem desses *tokens*, as expressões regulares que os definem e uma breve descrição a respeito de cada um.

Token	Expressão Regular	Descrição
Integer	[0-9]+	números inteiros não-negativos
Ident	[a-zA-Z_][a-zA-Z_0-9]*	identificadores que dão nome a índices, constantes e estruturas neurais
String	"([^\"]+ \"\\\")**	strings envoltas em aspas duplas
Newline	\n	quebra de linha
SingleLineComment	//.*	comentários de uma linha
MultiLineComment	/*(. \\n)*?\n*/	comentários multi-linha

Tabela 4.4: Demais *tokens* presentes na linguagem do SATyrus2

O resultado da análise léxica é uma lista de *tokens* que serão inspecionados pelo Parser durante o processo de análise sintática. Caso o compilador encontre caracteres ou grupos de caracteres inválidos, a análise léxica termina e uma exceção é disparada. O erro é posteriormente reportado pelo módulo de Interface com o usuário.

4.2.2 Análise sintática

Uma vez completa a análise léxica, a lista de *tokens* resultante é inspecionada pelo Parser, que busca por padrões válidos de *tokens*. Declarações de constantes, operações aritméticas e especificações de restrições de integridade e otimalidade são exemplos de padrões de *tokens* válidos dentro da linguagem SATish. O papel do analisador sintático é assegurar que todo o arquivo de modelagem fornecido pelo usuário - e mais especificamente, a lista de *tokens* gerada a partir desse arquivo - está em conformidade com as regras de formação da gramática definida pela linguagem SATish.²

Além de verificar a correção sintática dos arquivos de entrada, o Parser também é responsável por gerar a formulação lógica correspondente às restrições contidas na modelagem escrita pelo usuário do compilador. Esse processo será detalhado nos parágrafos a seguir.

A Listagem 4.7 apresenta um exemplo de restrição de integridade modelada em linguagem SATish:

```
intgroup int1 :  
  forall{i , j} : i in (1,2) , j in (1,2) :  
    A[i]  $\rightarrow$  B[i][j];
```

Listagem 4.7: Exemplo de restrição integridade modelada em linguagem SATish.

O primeiro passo realizado pelo compilador durante o processo de geração da LMS é converter a fórmula presente na restrição para a Forma Normal Conjuntiva (FNC), a partir da qual é possível se aplicar o mapeamento entre fórmulas e expressões algébricas definido na Seção 2.2.1. A conversão é ilustrada na Figura 4.3.

$$\begin{array}{l} A[i] \rightarrow B[i][j] \implies^1 \\ \neg(A[i] \rightarrow B[i][j]) \implies^2 \\ A[i] \wedge \neg B[i][j] \end{array}$$

Figura 4.3: Primeiro passo: negação da fórmula original; segundo passo: conversão para a FNC.

Após a conversão, os índices presentes na fórmula resultante devem ser *instanciados*. O processo de instanciação consiste em iterar os índices sobre o intervalo de valores definido no código da restrição. Por exemplo, os índices *i* e *j* presentes na restrição da Listagem 4.7 devem variar, cada um deles, dentro do intervalo [1, 2]. A instanciação é feita na ordem inversa à declaração dos índices. Assim, no exemplo

²O Apêndice A traz a especificação completa da linguagem SATish.

da Listagem 4.7, o índice j é instanciado antes do índice i , de maneira que um conjunto de fórmulas intermediárias é gerado com o índice j fixado, mas com o índice i ainda não instanciado, como ilustra a Tabela 4.5.

j		
1		2
i	$A[i] \wedge \neg B[i][1]$	$A[i] \wedge \neg B[i][2]$


Tabela 4.5: Fórmulas geradas com a instanciação do índice j

Após o índice j , é a vez do índice i variar dentro do intervalo definido na especificação da restrição. A Tabela 4.6 lista as fórmulas criadas após a instanciação do índice i .

j		
1		2
i	1	$A[1] \wedge \neg B[1][1]$
	2	$A[1] \wedge \neg B[1][2]$
i	1	$A[2] \wedge \neg B[2][1]$
	2	$A[2] \wedge \neg B[2][2]$

Tabela 4.6: Fórmulas geradas com a instanciação do índice i

A fórmula resultante de uma determinada restrição depende do quantificador lógico usado na especificação da mesma. O quantificador **forall**, utilizado na restrição da Listagem 4.7, deve ser interpretado da seguinte maneira: *para todos* os possíveis valores de i e de j , a fórmula presente na restrição deve ser válida. Dessa forma, **forall** implica uma *conjunção* das subfórmulas geradas nos passos de instanciação de índices. O quantificador **exists**, por outro lado, exige que *pelo menos uma* das subfórmulas geradas durante o processo de instanciação dos índices i e j seja satisfeita. O quantificador **exists**, portanto, implicaria em uma *disjunção* das subfórmulas da Tabela 4.6, caso fosse utilizado na restrição da Listagem 4.7.

Como consequência da utilização do quantificador **forall** na restrição da Listagem 4.7, a fórmula resultante, ilustrada na Figura 4.4, é uma conjunção das subfórmulas geradas durante o processo de instanciação dos índices i e j : 

$$(A[1] \wedge \neg B[1][1]) \vee (A[1] \wedge \neg B[1][2]) \vee (A[2] \wedge \neg B[2][1]) \vee (A[2] \wedge \neg B[2][2])$$

Figura 4.4: Fórmula resultante da restrição da Listagem 4.7.

SATyrus2 é um compilador de um passo. Ao contrário dos compiladores de dois passos, compiladores de um passo percorrem uma única vez o código fonte fornecido como entrada. No SATyrus2, a análise da correção sintática do código fonte é intercalada com a própria geração do código objeto resultante do processo de compilação. Em outras palavras, os erros de sintaxe, de contexto e outros possíveis erros de modelagem são detectados, caso existam, *durante* o procedimento de

geração de fórmulas lógicas apresentado nos parágrafos anteriores. São vários os exemplos de erros que podem ser cometidos pelo usuário durante a escrita de um modelo SATish: referências a constantes e índices indefinidos, utilização de índices que ultrapassam as dimensões estabelecidas para uma dada estrutura, omissão de símbolos característicos da linguagem (por exemplo, o sinal de ponto e vírgula que separa duas expressões) etc. O Apêndice B deve ser consultado como referência para todos possíveis erros de compilação retornados pelo SATyrus2.

Em caso de ausência de erros, a repetição do processo de instanciação de índices, restrição após restrição, dá origem a uma lista de fórmulas que representam, cada uma delas, uma restrição de integridade ou otimalidade presente na modelagem inicial escrita pelo usuário. Essa lista de fórmulas é o resultado da análise sintática, e será utilizada na fase seguinte do processo de compilação, a geração da função de energia, procedimento no qual entra em ação o mapeamento entre fórmulas e expressões algébricas definido no Capítulo 2.

4.2.3 Geração da função de energia

De acordo com as regras de mapeamento estabelecidas na Seção 2.2.1, a cada fórmula lógica gerada durante a análise sintática, corresponde uma expressão algébrica que comporá, junto com as demais, o corpo da função de energia.

As expressões algébricas que compõem a função de energia são obtidas uma a uma por meio da aplicação direta das regras de mapeamento estabelecidas na Seção 2.2.1. A linguagem SATish permite ao usuário especificar constantes multiplicativas em restrições de integridade ou otimalidade. Desse modo, tais restrições passam a ter penalidades com valores numéricos mais altos dentro da função de energia, o que faz com que seja menos vantajoso violá-las. O exemplo da Listagem 4.8 traz uma restrição de integridade que tem seu peso aumentado em 50 vezes.

```
intgroup int0 :
  forall{i,j}: i in (1,n), j in (1,n) :
    50 (A[i]  $\rightarrow$  B[i][j] or C[i]) ;
```

Listagem 4.8: Exemplo de restrição de integridade multiplicada por uma constante.

As restrições de integridade e otimalidade também são associadas a penalidades definidas pelo usuário. Tais penalidades são atribuídas às restrições mediante a criação de *grupos de restrições*. Em uma modelagem, todas as restrições devem necessariamente pertencer a um desses grupos. Estes definem conjuntos (possivelmente unitários) de restrições que possuem a mesma relevância dentro da modelagem original. Quanto mais relevante é uma restrição, mais alto é o valor numérico da

expressão algébrica correspondente dentro da função de energia, e por consequência, menos vantajoso é violar essa restrição.

Os grupos de restrições são definidos por meio das palavras chaves **intgroup** e **optgroup** (no caso de restrições de integridade ou otimalidade, respectivamente), seguidas pelo identificador que dá nome ao grupo. Por exemplo, a restrição definida na Figura 4.8 pertence ao grupo **int0**.

As penalidades associadas a cada grupo de restrições são calculadas de acordo com as regras definidas na Seção 2.2.3. Para um determinado grupo (G) de restrições, o cálculo da penalidade associada a G leva em consideração o número de cláusulas presentes nos grupos de restrições de nível mais baixo que G . A contagem das cláusulas é realizada automaticamente pelo compilador.

Calculados os valores numéricos das penalidades, o compilador divide a lista de fórmulas geradas durante a análise sintática em sublistas que correspondem uma a uma a cada um dos grupos distintos de restrições. Uma vez aplicado o mapeamento entre fórmulas e expressões algébricas a cada uma das fórmulas presentes em todas as sublistas, resta ao compilador multiplicar cada expressão algébrica pelo valor numérico da penalidade correspondente. A Figura 4.5 ilustra esse processo:

$$\text{Restrições} \begin{Bmatrix} R_1 \\ R_2 \\ \vdots \\ R_n \end{Bmatrix} \rightarrow \text{Fórmulas} \begin{Bmatrix} F_1 \\ F_2 \\ \vdots \\ F_n \end{Bmatrix} \rightarrow \text{Expressões} \begin{Bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{Bmatrix} \rightarrow \sum_{i=1}^n P_i * E_i$$

Figura 4.5: Processo de obtenção da função de energia (P_i denota o valor da penalidade no nível de restrições i).

A função de energia corresponde à somatória de todas as expressões algébricas produzidas pelo método representado na Figura 4.5. Uma vez construída, a função de energia é armazenada em memória pelo compilador para que possa ser utilizada adiante pelo módulo de Interface com softwares de otimização. A explicação do processo de geração de modelos AMPL e Xpress requer um maior detalhamento desses softwares, e portanto, uma seção à parte.

4.3 Interface com softwares de otimização

A geração de modelos de AMPL e Xpress é o último passo do processo de compilação realizado pelo SATyrus2. Este processo consiste em adaptar a função de energia à linguagem do software de otimização escolhido pelo usuário. Também é dever do módulo de Interface com softwares de otimização garantir que as exigências listadas na página 34 sejam cumpridas, a saber:

- as variáveis da função de energia devem ser definidas como variáveis binárias;
- a função de energia deve ser minimizada;
- uma estratégia adequada de busca da solução ótima deve ser empregada pelo software de otimização;
- ao final do procedimento de minimização da função de energia, a solução encontrada deve ser exibida para o usuário do compilador.

O módulo de Interface com softwares de otimização é dividido em tantos submódulos quantos forem os softwares de otimização suportados pelo SATyrus2. Atualmente, o código fonte do SATyrus2 possui dois destes submódulos: o primeiro faz interface com AMPL; o segundo, com Xpress. Por serem inteiramente independentes, outros destes submódulos podem ser acrescentados ao código fonte do compilador, aumentando assim o número de softwares com os quais o SATyrus2 pode interagir.

Os processos de geração e execução de modelos AMPL e Xpress são detalhados nas Seções 4.3.1 e 4.3.2, respectivamente. Os exemplos descritos nas seções a seguir têm como base a modelagem da Listagem 4.9, um modelo trivial de TSP com três cidades (uma delas copiada) escrito em linguagem SATish.

```

n=3;
pos(n,n);
dist(n,n);

pos = [ 1,1: 1; 3,3: 1];

dist = [
    1,1: 0; 2,1: 4; 3,1: 0;
    1,2: 4; 2,2: 0; 3,2: 4;
    1,3: 0; 2,3: 4; 3,3: 0
];

intgroup tour:
    forall{i,j} where i in (1,n), j in (1,n):
        pos[i][j];

intgroup wta:
    forall{i,j,k}
        where i in (1,n), j in (1,n), k in (1,n) and i!=k:

```

```

    pos[i][j] -> not pos[k][j];

intgroup wta:
    forall{i,j,l}
        where i in (1,n), j in (1,n), l in (1,n) and j!=l:
            pos[i][j] -> not pos[i][l];

optgroup cost:
    forall{i,j,k}
        where i in (1,n), j in (2,n), k in (1,n) and i!=k:
            dist[i][k] (pos[i][j] and pos[k][j-1]);

optgroup cost:
    forall{i,j,k}
        where i in (1,n), j in (1,n-1), k in (1,n) and i!=k:
            dist[i][k] (pos[i][j] and pos[k][j+1]);

penalties:
    cost level 0;
    tour level 1;
    wta level 2;

```

Listagem 4.9: TSP de duas cidades modelado em SATish.

4.3.1 AMPL

AMPL é uma plataforma de modelagem algébrica destinada a problemas lineares e não-lineares de otimização, sejam eles discretos ou contínuos [2]. A plataforma AMPL consiste em dois fatores: uma linguagem declarativa (também denominada AMPL) disponibilizada ao usuário para a elaboração de modelos computacionais, e um ambiente de comandos interativo destinado à configuração e resolução de problemas de programação matemática. Adicionalmente, a plataforma AMPL permite que diversos resolvedores matemáticos estejam disponíveis ao usuário, que pode optar pelo resolvedor mais apropriado para um certo tipo de problema. Uma vez encontrada a melhor solução possível para um dado problema, AMPL exibe a solução para o usuário como resultado final da computação.

Cabe ao SATyrus2 transformar modelos escritos em linguagens SATish em arquivos de modelagem válidos em linguagem AMPL. O usuário do SATyrus2, por sua vez, é responsável por coletar os arquivos gerados pelo compilador e utilizá-los como parâmetros de entrada no ambiente AMPL.

O usuário do SATyrus2 pode escolher entre dois formatos de saída: modelos AMPL e modelos Xpress, gerados a partir de um arquivo qualquer escrito em linguagem SATish ³. Caso escolha o formato AMPL, dois arquivos serão escritos pelo compilador:

- um arquivo de controle, de extensão *.in*, que contém uma série de instruções destinadas ao compilador AMPL. Essas instruções garantem o cumprimento das duas últimas exigências listadas na página 43;
- um modelo de otimização, de extensão *.mod* e escrito em linguagem AMPL, que contém as variáveis e a função de energia geradas pelo SATyrus2 a partir do problema original.

AMPL não resolve diretamente os problemas de otimização: softwares resolvidores externos são utilizados na busca por uma solução. AMPL invoca os resolvidores como processos separados e se comunica com eles através da leitura e escrita de arquivos. Os arquivos têm nomes da forma *stub.sufixo*; o sufixo é automaticamente escolhido pelo compilador AMPL. Antes de invocar um resolvidor, AMPL escreve um arquivo chamado *stub.nl*, que contém uma descrição do problema a ser resolvido. O compilador AMPL invoca o resolvidor passando como argumento o arquivo *stub* construído no passo anterior. A partir desse instante, AMPL aguarda que o resolvidor escreva um arquivo chamado *stub.sol*, contendo uma mensagem de término e a solução encontrada para o problema. [14]

O fluxo de informação entre SATyrus2, AMPL e resolvidores está ilustrado na Figura 4.6.

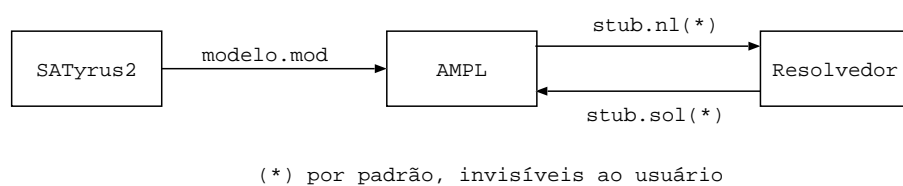


Figura 4.6: Integração entre SATyrus2, AMPL e Resolvedor.

Resolvedores

AMPL oferece uma variedade de resolvidores que podem ser utilizados pelo usuário para otimizar problemas matemáticos de naturezas diversas: problemas lineares, não-lineares, convexos etc. A lista de resolvidores disponíveis no ambiente AMPL se encontra em [13].

³As opções de compilação do SATyrus2 estão listadas no Apêndice B.

Os problemas de otimização abordados neste trabalho são problemas de variáveis inteiras e função de energia não-convexa e não-linear. Foi necessário encontrar, dentre os resolvidores disponíveis, um software capaz de lidar com problemas dessa natureza. Bonmin [16] foi o resolvidor escolhido.

Bonmin é um software experimental projetado para resolver problemas de programação não-linear inteira mista. Bonmin é capaz de resolver problemas do tipo:

determine

$$\min f(x)$$

sujeito a:

$$g_L \leq g(x) \leq g_U$$

$$x_L \leq x \leq x_U$$

$$\forall i \in I : x_i \in \mathbb{Z}, \text{ e}$$

$$\forall i \notin I : x_i \in \mathbb{R}$$

onde:

$$f : \mathbb{R}^n \rightarrow \mathbb{R}$$

$$g : \mathbb{R}^n \rightarrow \mathbb{R}^m, \text{ e}$$

$$I \subseteq \{1, \dots, n\}$$

dentre os quais se encaixam os problemas matemáticos inteiros e não-convexos criados pelo SATyrus2. Os algoritmos utilizados por Bonmin são exatos quando as funções f e g são convexas; caso f e g não o sejam, o resolvidor utiliza heurísticas na busca por uma solução ótima.

Exemplo

A Figura 4.7 apresenta o modelo AMPL gerado a partir da instância do TSP transcrita na Listagem 4.9. O modelo é dividido em duas seções: a primeira contém a especificação das variáveis presentes no problema; a segunda possui a função de energia a ser minimizada.

Todas as variáveis são declaradas como binárias através da utilização da palavra chave **binary** logo após o identificador de cada uma das variáveis. Isso garante que a primeira exigência listada na página 43 seja cumprida. A segunda exigência (minimização da função de energia) é satisfeita pela palavra chave **minimize**, situada imediatamente antes do identificador que dá nome à função de energia. As linhas iniciadas pelo caractere ‘#’ são comentários inseridos pelo SATyrus2.

A Figura 4.8 ilustra o arquivo de controle gerado pelo SATyrus2 a partir da instância do TSP definida na Listagem 4.9:

```

# Variables:
var pos_3_2 binary;
var pos_3_1 binary;
var pos_1_2 binary;
var pos_1_3 binary;
var pos_2_3 binary;
var pos_2_2 binary;
var pos_2_1 binary;

# Energy Function:
minimize f: 48.001000 * (((1 - 1) + (1 - pos_1_2) + (1 - pos_1_3) +
(1 - pos_2_1) + (1 - pos_2_2) + (1 - pos_2_3) + (1 - pos_3_1) +
(1 - pos_3_2) + (1 - 1))) + 480.011000 * (((pos_2_1 * 1) + (pos_3_1 * 1) +
(pos_2_2 * pos_1_2) + (pos_3_2 * pos_1_2) + (pos_2_3 * pos_1_3) +
(1 * pos_1_3) + (1 * pos_2_1) + (pos_3_1 * pos_2_1) + (pos_1_2 * pos_2_2) +
(pos_3_2 * pos_2_2) + (pos_1_3 * pos_2_3) + (1 * pos_2_3) + (1 * pos_3_1) +
(pos_2_1 * pos_3_1) + (pos_1_2 * pos_3_2) + (pos_2_2 * pos_3_2) +
(pos_1_3 * 1) + (pos_2_3 * 1))) + 480.011000 * (((pos_1_2 * 1) +
(pos_1_3 * 1) + (1 * pos_1_2) + (pos_1_3 * pos_1_2) + (1 * pos_1_3) +
(pos_1_2 * pos_1_3) + (pos_2_2 * pos_2_1) + (pos_2_3 * pos_2_1) +
(pos_2_1 * pos_2_2) + (pos_2_3 * pos_2_2) + (pos_2_1 * pos_2_3) +
(pos_2_2 * pos_2_3) + (pos_3_2 * pos_3_1) + (1 * pos_3_1) +
(pos_3_1 * pos_3_2) + (1 * pos_3_2) + (pos_3_1 * 1) + (pos_3_2 * 1))) +
1.000000 * (((4 * pos_1_2 * pos_2_1) + (0 * pos_1_2 * pos_3_1) +
(4 * pos_1_3 * pos_2_2) + (0 * pos_1_3 * pos_3_2) + (4 * pos_2_2 * 1) +
(4 * pos_2_2 * pos_3_1) + (4 * pos_2_3 * pos_1_2) + (4 * pos_2_3 * pos_3_2) +
(0 * pos_3_2 * 1) + (4 * pos_3_2 * pos_2_1) + (0 * 1 * pos_1_2) +
(4 * 1 * pos_2_2))) + 1.000000 * (((4 * 1 * pos_2_2) + (0 * 1 * pos_3_2) +
(4 * pos_1_2 * pos_2_3) + (0 * pos_1_2 * 1) + (4 * pos_2_1 * pos_1_2) +
(4 * pos_2_1 * pos_3_2) + (4 * pos_2_2 * pos_1_3) + (4 * pos_2_2 * 1) +
(0 * pos_3_1 * pos_1_2) + (4 * pos_3_1 * pos_2_2) + (0 * pos_3_2 * pos_1_3) +
(4 * pos_3_2 * pos_2_3)));

```

Figura 4.7: Modelo AMPL gerado a partir do código da Listagem 4.9.

```

model tsp.mod;
option solver bonmin;
solve;
display {i in 1.._nvars} (_varname[i], _var[i]);

```

Figura 4.8: Arquivo AMPL de controle para o código da Listagem 4.9.

A primeira linha do arquivo 4.8 especifica o nome do modelo a ser compilado por AMPL. O resolvidor que será utilizado na resolução do modelo é especificado na segunda linha. No exemplo da acima, Bonmin foi o resolvidor escolhido.

A terceira linha do arquivo de controle contém o comando **solve**, que instrui o compilador AMPL a invocar o resolvidor indicado na linha 2 para que o procedimento de resolução do problema tenha início. A última linha do arquivo de controle faz com que AMPL imprima o valor final das variáveis quando uma solução for atingida pelo resolvidor.

O formato exato da saída gerada por AMPL ao término do processo de busca por uma solução depende do resolvidor escolhido pelo usuário. No entanto, duas informações estão sempre presentes: o estado final das variáveis presentes no problema e o valor alcançado pela função de energia (f) na solução proposta.

A Figura 4.9 ilustra o formato da saída gerada por AMPL quando os arquivos das Figuras 4.7 e 4.8 são utilizados como parâmetros de entrada. As informações relevantes na saída gerada por AMPL para este exemplo são:

- os valores pelos quais passou a função de energia durante o processo de busca por uma solução ótima, listados na coluna *Obj*;
- o valor final da função de energia, impresso na linha: “*Search completed - best objective 304.006*”;
- o estado final das variáveis do problema, listados nas últimas linhas do resultado.

```

bonmin:

*****
This program contains Ipopt, a library for large-scale nonlinear optimization.
Ipopt is released as open source code under the Common Public License (CPL).
For more information visit http://projects.coin-or.org/Ipopt
*****

NLP0012I
      Num      Status      Obj      It      time
NLP0013I      1      OPT      304.0058871987207      9      0.012001
NLP0012I
      Num      Status      Obj      It      time
NLP0013I      1      OPT      304.006      0      0
Cbc0004I Integer solution of 304.006 found after 0 iterations and 0 nodes
(0.00 seconds)
Cbc0001I Search completed - best objective 304.006, took 0 iterations and
0 nodes (0.00 seconds)
Cbc0035I Maximum depth 0, 0 variables fixed on reduced cost

      "Finished"

bonmin: Optimal
: _varname[i] _var[i]      :=
1  pos_3_2      0
2  pos_3_1      0
3  pos_1_2      0
4  pos_1_3      0
5  pos_2_3      0
6  pos_2_2      1
7  pos_2_1      0
;

```

Figura 4.9: Resultado da execução do modelo da Figura 4.7.

4.3.2 Xpress

Xpress é um ambiente de modelagem matemática destinado à resolução de diversos tipos de problemas de otimização. A plataforma Xpress está atrelada a Mosel: uma linguagem simultaneamente imperativa e declarativa projetada para a especificação e resolução de problemas matemáticos.

O usuário do SATyrus2 pode solicitar a geração de modelos Xpress através das opções fornecidas pelo compilador por meio do módulo de Interface com o usuário. Ao contrário do formato AMPL, descrito na seção anterior, o formato Xpress é composto por um único arquivo; este é escrito em linguagem Mosel e possui extensão *.mos*. Não há um arquivo de controle, uma vez que todas as instruções de escolha do resolvidor, resolução da função de energia e exibição da solução atingida fazem parte da linguagem Mosel, como descrito nas seções a seguir.

Resolvedores

Ao contrário de AMPL, Xpress não utiliza resolvedores externos à plataforma. Em Xpress, os resolvedores são módulos que fazem parte do código fonte do ambiente, e que podem ser estendidos pelo usuário. Estão disponíveis módulos de resolução de problemas lineares e inteiros, problemas não-lineares e também módulos que permitem a formulação e resolução de problemas de programação estocástica [17].

O módulo de resolução de problemas não-lineares, denominado **mmxslp**, é o resolvidor utilizado pelo SATyrus2.

Exemplo

Ao contrário do formato AMPL, os modelos Mosel construídos pelo SATyrus2 não são divididos em dois arquivos. Em Mosel, o resolvidor a ser utilizado na busca por uma solução é especificado dentro do arquivo de modelagem. Também estão dentro do modelo os comandos que instruem o ambiente Xpress a exibir o resultado final da computação.

A Figura 4.10 apresenta o arquivo Mosel gerado pelo SATyrus2 a partir do modelo da Listagem 4.9. Assim como nos modelos AMPL, as variáveis são declaradas como variáveis binárias, desta vez em conformidade com a sintaxe da linguagem Mosel. O resolvidor a ser utilizado na minimização da função de energia é especificado na terceira linha, e os comandos de impressão do estado final das variáveis se encontram na última seção do arquivo.

```

model SATModel

uses "mmxslp"

declarations
pos_3_2: mpvar; pos_3_1: mpvar
pos_1_2: mpvar; pos_1_3: mpvar
pos_2_3: mpvar; pos_2_2: mpvar
pos_2_1: mpvar

objdef: mpvar
end-declarations

pos_3_2 is_binary; pos_3_1 is_binary
pos_1_2 is_binary; pos_1_3 is_binary
pos_2_3 is_binary; pos_2_2 is_binary
pos_2_1 is_binary

energy_function := 48.001000 * (((1 - 1) + (1 - pos_1_2) + (1 - pos_1_3) +
(1 - pos_2_1) + (1 - pos_2_2) + (1 - pos_2_3) + (1 - pos_3_1) +
(1 - pos_3_2) + (1 - 1))) + 480.011000 * (((pos_2_1 * 1) + (pos_3_1 * 1) +
(pos_2_2 * pos_1_2) + (pos_3_2 * pos_1_2) + (pos_2_3 * pos_1_3) +
(1 * pos_1_3) + (1 * pos_2_1) + (pos_3_1 * pos_2_1) + (pos_1_2 * pos_2_2) +
(pos_3_2 * pos_2_2) + (pos_1_3 * pos_2_3) + (1 * pos_2_3) + (1 * pos_3_1) +
(pos_2_1 * pos_3_1) + (pos_1_2 * pos_3_2) + (pos_2_2 * pos_3_2) +
(pos_1_3 * 1) + (pos_2_3 * 1))) + 480.011000 * (((pos_1_2 * 1) +
(pos_1_3 * 1) + (1 * pos_1_2) + (pos_1_3 * pos_1_2) + (1 * pos_1_3) +
(pos_1_2 * pos_1_3) + (pos_2_2 * pos_2_1) + (pos_2_3 * pos_2_1) +
(pos_2_1 * pos_2_2) + (pos_2_3 * pos_2_2) + (pos_2_1 * pos_2_3) +
(pos_2_2 * pos_2_3) + (pos_3_2 * pos_3_1) + (1 * pos_3_1) +
(pos_3_1 * pos_3_2) + (1 * pos_3_2) + (pos_3_1 * 1) + (pos_3_2 * 1))) +
1.000000 * (((4 * pos_1_2 * pos_2_1) + (0 * pos_1_2 * pos_3_1) +
(4 * pos_1_3 * pos_2_2) + (0 * pos_1_3 * pos_3_2) + (4 * pos_2_2 * 1) +
(4 * pos_2_2 * pos_3_1) + (4 * pos_2_3 * pos_1_2) + (4 * pos_2_3 * pos_3_2) +
(0 * pos_3_2 * 1) + (4 * pos_3_2 * pos_2_1) + (0 * 1 * pos_1_2) +
(4 * 1 * pos_2_2))) + 1.000000 * (((4 * 1 * pos_2_2) + (0 * 1 * pos_3_2) +
(4 * pos_1_2 * pos_2_3) + (0 * pos_1_2 * 1) + (4 * pos_2_1 * pos_1_2) +
(4 * pos_2_1 * pos_3_2) + (4 * pos_2_2 * pos_1_3) + (4 * pos_2_2 * 1) +
(0 * pos_3_1 * pos_1_2) + (4 * pos_3_1 * pos_2_2) + (0 * pos_3_2 * pos_1_3) +
(4 * pos_3_2 * pos_2_3)))
objdef = energy_function

SLPloadprob(objdef)
SLPminimize

writeln("Objective: ", getobjval)
writeln("pos_3_2 = ", getsol(pos_3_2)); writeln("pos_3_1 = ", getsol(pos_3_1))
writeln("pos_1_2 = ", getsol(pos_1_2)); writeln("pos_1_3 = ", getsol(pos_1_3))
writeln("pos_2_3 = ", getsol(pos_2_3)); writeln("pos_2_2 = ", getsol(pos_2_2))
writeln("pos_2_1 = ", getsol(pos_2_1))

writeln

end-model

```

Figura 4.10: Modelo Mosel gerado a partir do código da Listagem 4.9.

O aspecto do resultado final exibido por Xpress é configurável pelo usuário, que pode formatá-lo através dos comandos de impressão fornecidos pela linguagem Mosel. SATyrus2 constrói modelos Mosel que exibem, ao final da computação, os valores finais alcançados pela função de energia e por todas as variáveis que compõem o problema. O resultado da execução do modelo 4.10 está ilustrado na Figura 4.11:

```
$ mosel -c "exec tsp"
Objective: 304.0000
pos_3_2 = 0
pos_3_1 = 0
pos_1_2 = 0
pos_1_3 = 0
pos_2_3 = 0
pos_2_2 = 1
pos_2_1 = 0

Xpress-Mosel v2.4.0
(c) Copyright Fair Isaac Corporation 2008
>exec out
Returned value: 0
>
Exiting.
```

Figura 4.11: Resultado da execução do modelo da Figura 4.10.

AMPL e Xpress são as duas plataformas de resolução de problemas disponíveis ao usuário do SATyrus2. O Capítulo 5 oferece uma análise dos resultados da integração dessas plataformas com o compilador SATyrus2, por meio da elaboração de modelagens maiores e mais complexas.

Capítulo 5

Especificações de Raciocínio Lógico

Neste capítulo, o desempenho e o comportamento da plataforma SATyrus2 são analisados à luz de problemas maiores e mais complexos. É apresentada a ARQ-PROP II, um sistema de inferência lógica que pode ser expresso em termos da linguagem SATish. Adicionalmente, os modelos AMPL e Xpress gerados pelo SATyrus2 a partir da ARQ-PROP II são apresentados e discutidos. Também são analisados o desempenho do compilador e os resultados da execução de ARQ-PROP II na plataforma SATyrus2.

Todos os testes descritos neste capítulo foram feitos com a seguinte configuração de hardware e sistema operacional:

- Processador 0: Intel(R) Pentium(R) Dual CPU T2390 @1.86GHz
- Processador 1: Intel(R) Pentium(R) Dual CPU T2390 @1.86GHz
- Memória: 2GB
- Sistema Operacional: GNU/Linux Kernel 2.6.31-20

5.1 ARQ-PROP II

A ARQ-PROP II é um sistema capaz de realizar raciocínio baseado em lógica proposicional, proposto por Lima em [28] [29]. A ARQ-PROP II é capaz de trabalhar tanto com conhecimento completo quanto incompleto, e pode ser modelada através da definição de conjuntos de variáveis proposicionais e restrições que provêm um raciocinador capaz de realizar inferências baseadas no Princípio da Resolução. Adicionalmente, para que se possa raciocinar com conhecimento incompleto, o mecanismo deve ser capaz de criar novas sentenças (cláusulas).

Os conjuntos de variáveis proposicionais e as restrições utilizadas na modelagem da ARQ-PROP II são apresentadas nas seções a seguir.

5.1.1 Estruturas

A ARQ-PROP II faz uso de 11 estruturas de variáveis, cada uma delas responsável por uma determinada tarefa do processo de inferência lógica. As duas estruturas principais, PROOF e CLCOMP, representam, respectivamente, a área de prova sobre a qual o processo de inferência é realizado e o conjunto de proposições que compõem o conhecimento sobre o qual se deseja trabalhar. Para que possam ser utilizadas pela ARQ-PROP II, as proposições devem ser representadas sob a forma de cláusulas.

Para um problema de dimensão d (que equivale a uma área de prova que comporte ao máximo d cláusulas), CLCOMP possui $d \times d \times 2$ variáveis: estas são utilizadas como referências para os literais presentes na base de cláusulas considerada. A variável $CLCOMP_{ijk}$ equivale ao j -ésimo literal da i -ésima cláusula da base de cláusulas, onde k representa o sinal do literal: $k = 1$ para literais negativos, e $k = 2$ para literais positivos.

A estrutura PROOF tem dimensão de $d \times d \times 2$ variáveis. Cada linha de PROOF representa uma cláusula: esta pode ser uma cópia da base de cláusulas ou fruto de um passo de resolução. $PROOF_{ijk}$ simboliza o j -ésimo literal (de sinal igual a k) que se encontra na i -ésima linha da área de prova.

A título de exemplo, considere a seguinte base de cláusulas: $\{p, (q \vee \neg r), \neg q\}$. Em primeiro lugar, deve-se indexar (arbitrariamente) todas as cláusulas e todos os literais presentes na base de cláusulas sobre a qual se deseja trabalhar. Portanto, sejam: p a cláusula número 1 (C_1), $(q \vee \neg r)$ a cláusula número 2 (C_2) e $\neg q$ a cláusula número 3 (C_3). Sejam também p o literal número 1 (l_1), q o literal número 2 (l_2) e r o literal número 3 (l_3). Para este exemplo, a configuração de CLCOMP é a seguinte:

+		
	+	-
	-	

Tabela 5.1: Exemplo de configuração de CLCOMP.

Na Tabela 5.1, C_1 se encontra na primeira linha. Essa cláusula é formada unicamente pelo literal l_1 , mapeado na primeira coluna da tabela. Os sinais ‘+’ e ‘-’, que indicam o sinal de determinado literal, representam a terceira dimensão da estrutura CLCOMP. As demais cláusulas, C_2 e C_3 , estão representadas, respectivamente, na segunda e terceira linhas da Tabela 5.1.

A ARQ-PROP II funciona através de um mecanismo que copia para a área de prova as cláusulas presentes na estrutura CLCOMP. Por meio do Princípio da

Resolução lógica, no qual literais de sinais diferentes são cancelados em cláusulas distintas, é possível derivar novas cláusulas e expandir o conhecimento inicial. Para completar tal tarefa, a ARQ-PROP II inclui estruturas adicionais. São elas:

- IN ($d \times 1$): indica se uma linha faz parte da prova ou não;
- CB ($d \times 1$): indica se uma linha da área de prova foi copiada da base de cláusulas ou não;
- RES ($d \times 1$): indica se uma linha da área de prova é fruto de um passo de resolução;
- EMPTY ($d \times 1$): indica se uma linha da área de prova é uma cláusula vazia;
- CBMAP ($d \times d$): mapeia linhas da área de prova às cláusulas presentes na base de cláusulas das quais derivam;
- PARENT ($d \times d \times 2$): indica os pais (pai_1 e pai_2) de uma linha da área de prova que resulta de um passo de resolução;
- CANCEL ($d \times d$): indica qual proposição foi cancelada nas linhas da área de prova que resultam de passos de resolução;
- ORIG ($d \times d$): indica se uma cláusula pertence ao conhecimento original ou se é uma cláusula inventada.

Todas essas estruturas são utilizadas nas restrições apresentadas na próxima seção, que definem o mecanismo de inferência lógica da ARQ-PROP II.

5.1.2 Restrições

Existem duas maneiras de modelar a ARQ-PROP II. A *modelagem em alto nível* faz uso de restrições com existencial que dão origem a disjunções de cláusulas para as quais se deseja que apenas uma seja verdadeira em dado momento. No entanto, uma vez que o mapeamento proposto por Pinkas não inclui o operador de ou-exclusivo, a *modelagem anotada* da ARQ-PROP II, daqui em diante denominada apenas *modelagem*, utiliza restrições WTA para eliminar os existenciais e assegurar, ao mesmo tempo, a validade das disjunções exclusivas.

Todas as restrições que modelam a ARQ-PROP II são restrições de integridade; não há restrições de otimalidade na modelagem apresentada a seguir. As restrições da ARQ-PROP II são agrupadas em três níveis: as restrições do primeiro nível, denominado nível 1, correspondem às restrições das quais o existencial foi retirado. Estas são semelhantes às restrições de nível mais baixo presentes nas modelagens

do TSP e do problema da Coloração de Grafos, ambas apresentadas no Capítulo 2. As restrições do segundo nível, denominado nível 2, são aquelas que não possuem existenciais implícitos. Por fim, as restrições do nível mais alto são WTAs que garantem a validade do *ou-exclusivo* para as cláusulas geradas pelas restrições do primeiro nível.

As restrições que modelam a ARQ-PROP II são as seguintes:

1. Restrições das estruturas IN e PROOF:

- (a) Uma linha da área de prova é uma cópia da base de cláusulas ou fruto de um passo de resolução:

$$\forall i | 1 \leq i \leq d: IN_i \rightarrow CB_i \vee RES_i$$

Nível de penalidade: 2

- (b) Uma linha da área de prova que não seja uma cláusula vazia pode ser pai de outras linhas da área de prova:

$$\forall i, j, k | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq 2:$$

$$IN_i \wedge \neg EMPTY_i \rightarrow PARENT_{ijk}$$

Nível de penalidade: 1

- (c) Uma cláusula vazia não pode ser pai de outras linhas da área de prova:

$$\forall i, j, k | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq 2:$$

$$IN_i \wedge EMPTY_i \rightarrow \neg PARENT_{ijk}$$

Nível de penalidade: 2

2. Restrições de uma cláusula:

- (a) Uma linha da área de prova que é cópia de uma entrada na base de cláusulas deve estar associada a esta por meio de CBMAP:

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

Nível de penalidade: 1

- (b) 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 da base de cláusulas devem estar presentes na linha que representa i na área de prova:

- (a) $\forall i, j, k, s | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, 1 \leq s \leq 2:$

$$CBMAP_{ij} \wedge CLCOMP_{jks} \rightarrow PROOF_{iks}$$

Nível de penalidade: 2

- (b) $\forall i, j, k, s | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, 1 \leq s \leq 2:$

$$CBMAP_{ij} \wedge \neg CLCOMP_{jks} \rightarrow \neg PROOF_{iks}$$

Nível de penalidade: 2

4. Restrições dos passos de resolução:

- (a) Cada linha resultante de um passo de resolução deve ser fruto de duas outras linhas (denominadas *pais*):

$$\forall i, j, k, l, m | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, 1 \leq l \leq 2, 1 \leq m \leq 2; j \neq k, m \neq l:$$

$$RES_i \rightarrow (PARENT_{jil} \wedge PARENT_{kim})$$

Nível de penalidade: 1

- (b) Deve haver um par de literais cancelados 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:

- (a) As linhas envolvidas em um passo de resolução devem ter conter um membro do par cancelado:

$$\forall i, j, k, l, m, n | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, 1 \leq l \leq d, 1 \leq m \leq 2, 1 \leq n \leq 2; j \neq k, m \neq n:$$

$$PARENT_{ji1} \wedge PARENT_{ki2} \wedge PROOF_{jlm} \wedge PROOF_{kln} \rightarrow CANCEL_{ij}$$

Nível de penalidade: 2

- (b) Somente as linhas que são resultado de um passo de resolução têm pais:

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

Nível de penalidade: 2

- (c) Uma linha resultante de um passo de resolução deve fazer parte da prova:

$$\forall i, j, k | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq 2: PARENT_{jik} \rightarrow IN_i$$

Nível de penalidade: 2

- (d) O pai de uma linha da área de prova também deve formar parte da prova:

$$\forall i, j, k | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq 2: PARENT_{jik} \rightarrow IN_j$$

Nível de penalidade: 2

6. Restrições da composição do resolvente:

- (a) Copia ao resolvente todos os literais dos pais que não foram cancelados:

$$\forall i, j, k, l, s | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, 1 \leq l \leq 2, 1 \leq s \leq 2:$$

$$PARENT_{jil} \wedge PROOF_{jks} \wedge \neg CANCEL_{ik} \rightarrow PROOF_{iks}$$

Nível de penalidade: 2

- (b) Copia ao resolvente somente os literais que estavam nos pais:
 $\forall i, j, k, l, m, n, p | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, 1 \leq l \leq d, 1 \leq m \leq 2, 1 \leq n \leq 2, 1 \leq p \leq 2; j \neq k, m \neq n:$
 $PARENT_{jim} \wedge PARENT_{kin} \wedge \neg PROOF_{jlp} \neg PROOF_{klp} \rightarrow \neg PROOF_{ilp}$
 Nível de penalidade: 2

- (c) Não copia o par de literais cancelado:
 $\forall i, j, s | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq s \leq 2: CANCEL_{ij} \rightarrow \neg PROOF_{ijs}$
 Nível de penalidade: 2

7. Restrições da cláusula vazia:

- (a) Uma linha da área de prova é vazia caso não possua literais:
 $\forall i, j, s | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq s \leq 2: EMPTY_i \rightarrow \neg PROOF_{ijs}$
 Nível de penalidade: 2
- (b) Uma cláusula vazia deve ser resultado de um passo de resolução:
 $\forall i | 1 \leq i \leq d: EMPTY_i \rightarrow RES_i$
 Nível de penalidade: 2

8. Restrições WTA:

- (a) Um símbolo proposicional deve aparecer só uma vez por linha da área de prova:
 $\forall i, j, s, k | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq s \leq 2, 1 \leq k \leq 2; s \neq k:$
 $PROOF_{ijs} \rightarrow \neg PROOF_{ijk}$
- (b) Uma linha da área de prova tem só uma justificativa (é cópia da base de cláusulas ou resultado de um passo de resolução):
 $\forall i | 1 \leq i \leq d: CB_i \rightarrow \neg RES_i$
- (c) Apenas uma cláusula pode ser copiada da base de cláusulas para uma determinada linha da área de prova:
 $\forall i, j, k | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d; j \neq k:$
 $CBMAP_{ij} \rightarrow \neg CBMAP_{ik}$
- (d) Cada linha resultante de um passo de resolução possui apenas um pai_1 e um pai_2 :
 $\forall i, j, k, l | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d, 1 \leq l \leq 2; j \neq k:$
 $PARENT_{jil} \rightarrow \neg PARENT_{kil}$
- (e) Cada linha da área de prova pode ser pai de outra linha apenas uma vez:
 $\forall i, j, n, k, l | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq n \leq d, 1 \leq k \leq 2, 1 \leq l \leq 2; i \neq n:$
 $PARENT_{jik} \rightarrow \neg PARENT_{jnl}$

- (f) As linhas pais envolvidas em um passo de resolução devem ser diferentes:
 $\forall i, j, k, l, m | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq 2, 1 \leq l \leq 2, 1 \leq m \leq 2; j \neq k:$
 $PARENT_{jil} \rightarrow \neg PARENT_{kim}$
- (g) Apenas um literal pode ser cancelado por passo de resolução:
 $\forall i, j, k | 1 \leq i \leq d, 1 \leq j \leq d, 1 \leq k \leq d; j \neq k:$
 $CANCEL_{ij} \rightarrow \neg CANCEL_{ik}$

O código completo da ARQ-PROP II em linguagem SATish se encontra no Apêndice C.

5.1.3 Compilação

De modo semelhante às outras modelagens apresentadas ao longo deste trabalho (TSP e Coloração de Grafos), a compilação da ARQ-PROP II na plataforma SATyrus2 consiste em traduzir a especificação transcrita no Apêndice C em modelos que possam ser diretamente utilizados nas plataformas AMPL e Xpress.

Em virtude do grande número de variáveis presentes na especificação (há cerca de 300 variáveis na modelagem da seção anterior), o número de expressões algébricas que compõem a função de energia da ARQ-PROP II também é elevado. O tamanho da função de energia impossibilita uma ilustração completa dos modelos AMPL e Xpress que resultam da compilação da ARQ-PROP II; contudo, as figuras 5.1 e 5.2 apresentam o aspecto geral desses modelos, nos quais variáveis e termos da função de energia foram omitidos por questões de espaço.

Nos testes realizados, a compilação da modelagem da ARQ-PROP II demandou 3.22 segundos, tanto para versão AMPL quanto para a versão Mosel. No entanto, o tempo de compilação não é significativo frente ao tempo necessário para otimizar a função de energia. Este assunto será discutido na próxima seção.

5.2 Testes

Quando comparada às modelagens do TSP e do problema de Coloração de Grafos, o número de variáveis envolvidas faz com que a ARQ-PROP II represente um teste de maior porte para o SATyrus2. Executar a ARQ-PROP II consiste em alimentar a modelagem apresentada na Seção 5.1.2 com valores fixos de variáveis: estes constituirão os dados de entrada para o problema. Em especial, é necessário especificar a base de cláusulas que representará o conhecimento utilizado durante o processo de inferência. Considere, por exemplo, a seguinte base de cláusulas:

$$\phi = \{(p \leftarrow q), (p \leftarrow r), q\}$$

```

# Variables:
var PARENT_1_1_1 binary;
var PARENT_1_1_2 binary;
var CANCEL_1_3 binary;
var CANCEL_1_2 binary;
(...)
var PROOF_5_1_1 binary;
var PROOF_5_1_2 binary;
var CLCOMP_1_5_1 binary;
var CLCOMP_1_5_2 binary;

# Energy Function:
minimize f: 324.000001 * (((1 - CB_1) * (1 - RES_1) * IN_1) +
((1 - CB_2) * (1 - RES_2) * IN_2) + ((1 - CB_3) * (1 - RES_3) * IN_3) +
((1 - CB_4) * (1 - RES_4) * IN_4) + ((1 - CB_5) * (1 - RES_5) * IN_5) +
((1 - CB_6) * (1 - RES_6) * IN_6))) +
1.000000 * (((1 - PARENT_1_1_1) * IN_1 * (1 - EMPTY_1)) +
(...))
(CANCEL_6_3 * CANCEL_6_5) + (CANCEL_6_4 * CANCEL_6_5) +
(CANCEL_6_6 * CANCEL_6_5) + (CANCEL_6_1 * CANCEL_6_6) +
(CANCEL_6_2 * CANCEL_6_6) + (CANCEL_6_3 * CANCEL_6_6) +
(CANCEL_6_4 * CANCEL_6_6) + (CANCEL_6_5 * CANCEL_6_6));

```

Figura 5.1: Resumo do modelo AMPL para o código da ARQ-PROP II.

Deseja-se saber se a partir da teoria constituída pelo conjunto de proposições ϕ pode-se derivar a proposição p . Isso pode ser feito através da introdução de $\neg p$ na base de cláusulas ϕ : caso seja possível derivar uma cláusula vazia (\perp) por meio de um ou mais passos de resolução, a proposição p será verdadeira. De fato, este é o caso para a base de cláusulas ϕ , como fica demonstrado no silogismo abaixo:

$$\frac{\frac{(p \leftarrow q) \quad q}{p} \quad \neg p}{\perp}$$

A descoberta de uma cláusula vazia termina o processo de inferência, e atesta a validade da proposição p .

Para testar a modelagem da ARQ-PROP II, a negação da proposição p foi incluída na base de cláusulas ϕ , dando origem a um novo conjunto de cláusulas a ser considerado:

$$\phi' = \{(p \leftarrow q), (p \leftarrow r), q, \neg p\}$$

O teste consiste em determinar se as regras de inferência modeladas sob a forma de restrições na Seção 5.1.2 são capazes de derivar a cláusula vazia. A estrutura CLCOMP, que representa a base de cláusulas a ser considerada, foi preenchida de acordo com o conjunto de cláusulas ϕ' . Estruturas auxiliares também foram parcialmente preenchidas com informações adicionais: por exemplo, é necessário copiar a cláusula $\neg p$ para a área de prova, por meio da estrutura CBMAP, para que


```

model SATModel

uses "mmxslp"

declarations
PARENT_1_1_1: mpvar
PARENT_1_1_2: mpvar
CANCEL_1_3: mpvar
CANCEL_1_2: mpvar
(...)
PROOF_5_1_1: mpvar
PROOF_5_1_2: mpvar
CLCOMP_1_5_1: mpvar
CLCOMP_1_5_2: mpvar

objdef: mpvar
end-declarations

PARENT_1_1_1 is_binary
PARENT_1_1_2 is_binary
CANCEL_1_3 is_binary
CANCEL_1_2 is_binary
(...)
PROOF_5_1_1 is_binary
PROOF_5_1_2 is_binary
CLCOMP_1_5_1 is_binary
CLCOMP_1_5_2 is_binary

energy_function := 324.000001 * (((((1 - CB_1) * (1 - RES_1) * IN_1) +
((1 - CB_2) * (1 - RES_2) * IN_2) + ((1 - CB_3) * (1 - RES_3) * IN_3) +
((1 - CB_4) * (1 - RES_4) * IN_4) + ((1 - CB_5) * (1 - RES_5) * IN_5) +
((1 - CB_6) * (1 - RES_6) * IN_6))) +
1.000000 * (((((1 - PARENT_1_1_1) * IN_1 * (1 - EMPTY_1)) +
(...)
(CANCEL_6_3 * CANCEL_6_5) + (CANCEL_6_4 * CANCEL_6_5) +
(CANCEL_6_6 * CANCEL_6_5) + (CANCEL_6_1 * CANCEL_6_6) +
(CANCEL_6_2 * CANCEL_6_6) + (CANCEL_6_3 * CANCEL_6_6) +
(CANCEL_6_4 * CANCEL_6_6) + (CANCEL_6_5 * CANCEL_6_6)));

objdef = energy_function

SLPloadprob(objdef)
SLPminimize

writeln("Objective: ", getobjval)
writeln("PARENT_1_1_1 = ", getsol(PARENT_1_1_1))
writeln("PARENT_1_1_2 = ", getsol(PARENT_1_1_2))
writeln("CANCEL_1_3 = ", getsol(CANCEL_1_3))
writeln("CANCEL_1_2 = ", getsol(CANCEL_1_2))
(...)
writeln("PROOF_5_1_1 = ", getsol(PROOF_5_1_1))
writeln("PROOF_5_1_2 = ", getsol(PROOF_5_1_2))
writeln("CLCOMP_1_5_1 = ", getsol(CLCOMP_1_5_1))
writeln("CLCOMP_1_5_2 = ", getsol(CLCOMP_1_5_2))

writeln

end-model

```

Figura 5.2: Resumo do modelo Mosel (Xpress) para a ARQ-PROP II.

o processo de inferência tenha início.

Esta instância de teste foi realizada por Espinoza em [6], o trabalho que apresentou a implementação da primeira versão da plataforma SATyrus. A solução obtida pelo resolvidor neural da primeira plataforma está exposta na Figura 5.3. Na ilustração, os pontos de cor preta correspondem às variáveis fixadas como verdadeiras na modelagem, e que constituem os dados de entrada para o problema. Os pontos de cor cinza são as variáveis setadas como verdadeiras durante o processo de busca por uma solução. Os demais pontos, de cor branca, correspondem às variáveis de valor booleano falso.

A solução exposta na Figura 5.3 descreve a obtenção da cláusula vazia (primeira linha da área de prova) por meio de um processo de inferências que fez uso da primeira, terceira e quarta proposições presentes na base de cláusulas ϕ' . Inicialmente, a proposição $\neg p$ foi copiada para a área de provas (sexta linha de PROOF). A cláusula $(p \leftarrow q)$ (equivalente a $\neg q \vee p$ na FNC) foi copiada para quinta linha de PROOF, e através da utilização do Princípio da Resolução entre as duas cláusulas, a proposição $\neg q$ foi gerada e armazenada na quarta linha da área de prova. A resolução de $\neg q$ e a proposição q , copiada a partir da base de cláusulas, dá origem à cláusula vazia, representada pela primeira linha da estrutura PROOF.

Acreditava-se que a solução ilustrada na Figura 5.3 era o ótimo global da função de energia construída a partir das restrições que modelam a ARQ-PROP II. Contudo, isso não se mostrou verdadeiro. Os testes realizados neste trabalho encontraram uma configuração final alternativa e menos custosa que a solução apresentada na Figura 5.3. A configuração exposta na Figura 5.4, obtida após 1440 segundos de execução dentro do resolvidor Bonmin, representa um estado final inconsistente em relação ao resultado esperado para o processo de inferências. No entanto, esta configuração possui um custo final mais baixo que a solução apresentada por Espinoza em [6].

Dois motivos foram considerados como possíveis causas para esta situação indesejada. Em primeiro lugar, este resultado poderia ser causado por um ou mais erros na implementação da plataforma SATyrus2. Por outro lado, também foi preciso considerar a possibilidade de haver problemas no método de modelagem da ARQ-PROP II.

Para excluir a possibilidade de erro no compilador, foi realizada uma análise manual que levou em consideração ambas as soluções aqui apresentadas: a configuração ilustrada na Figura 5.3, que corresponde à solução desejada, e o estado final exposto na Figura 5.4. Era necessário determinar qual das soluções possuía um custo mais baixo. Caso a solução esperada para esta instância da ARQ-PROP II não se revelasse ótima, a modelagem conteria um problema.

A análise realizada foi um comparativo entre o número de cláusulas insatisfeitas

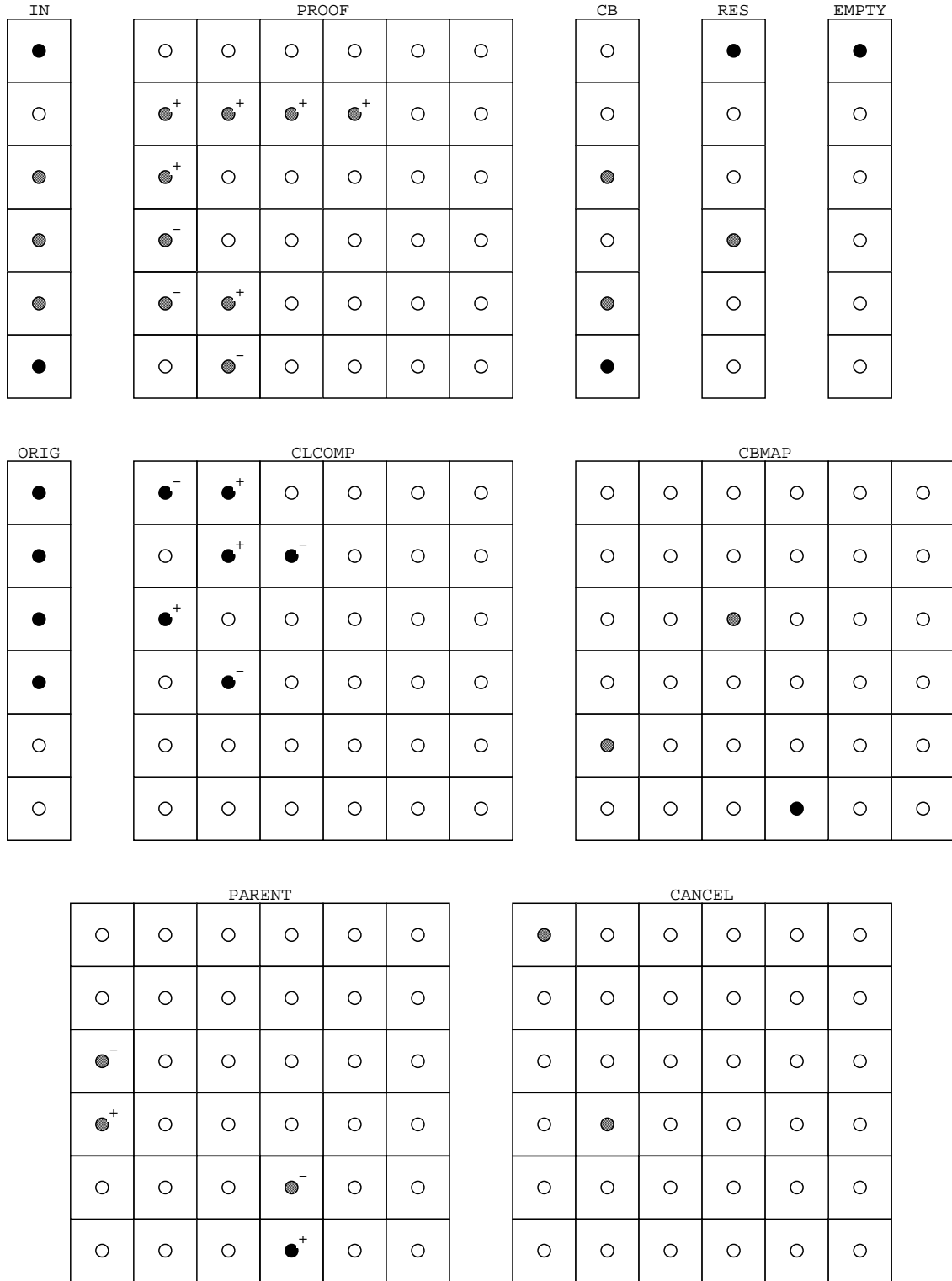


Figura 5.3: Resultado esperado para a base de cláusulas ϕ' .

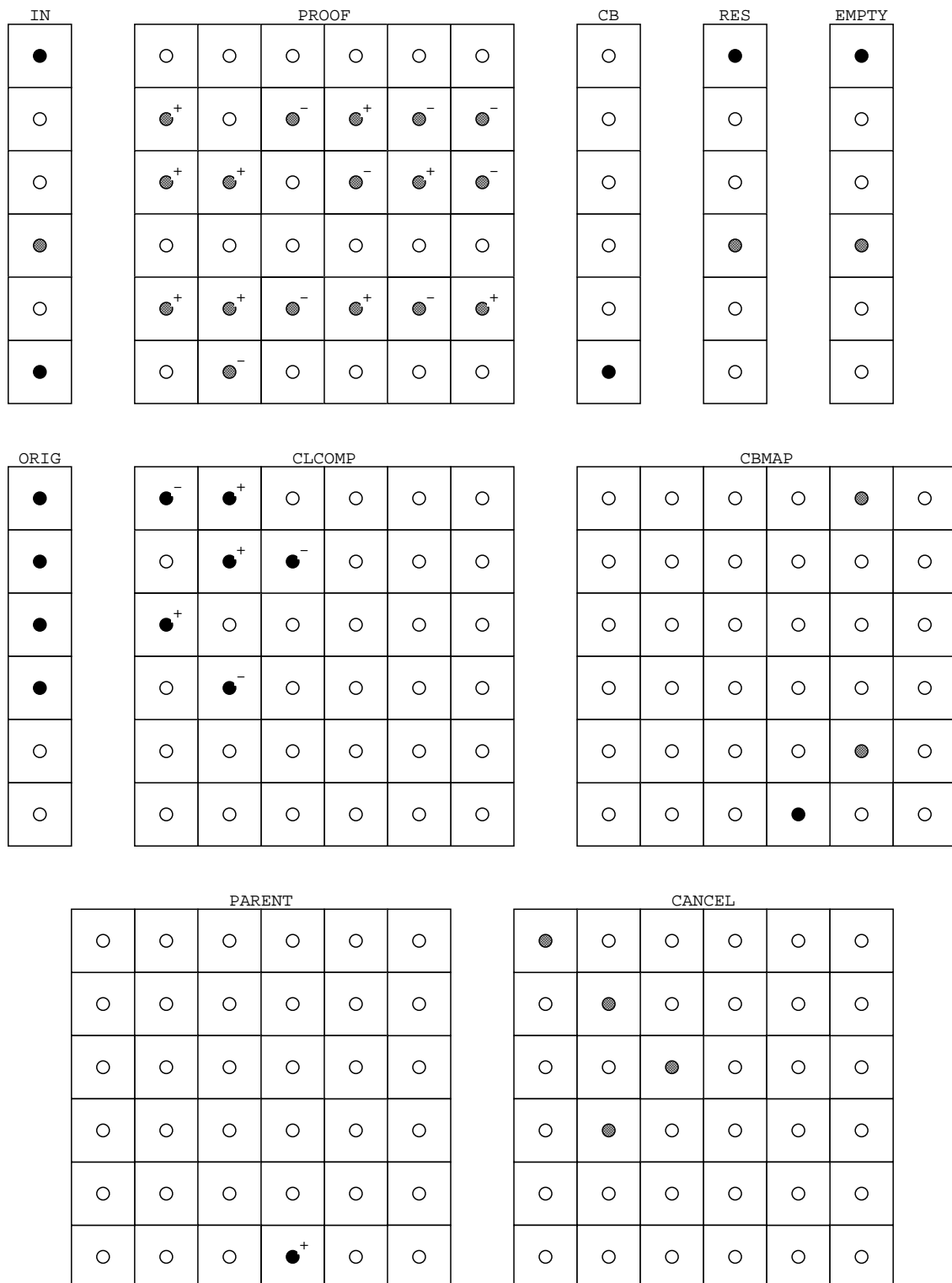


Figura 5.4: Resultado obtido pelo SATyrus2 para a base de cláusulas ϕ' .

nas configurações finais ilustradas nas figuras 5.3 e 5.4. Uma cláusula é insatisfeita quando a atribuição de valores booleanos às suas variáveis proposicionais fazem com que a cláusula seja avaliada como falsa. Cada cláusula insatisfeita corresponde a um acréscimo no valor final da função de energia de uma unidade multiplicada pelo valor numérico da penalidade associada à restrição à qual a cláusula pertence. Em outras palavras, para cada cláusula insatisfeita gerada por uma restrição de penalidade numérica p , a função de energia fica mais cara em p unidades.

A análise comparativa entre as soluções 5.3 e 5.4 constatou que existiam cláusulas insatisfeitas apenas nas restrições de nível 1, aquelas nas quais um existencial implícito é simplificado em uma série de conjunções. Por exemplo, considere a restrição 1(b), presente na modelagem da Seção 5.1.2:

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

Para a configuração ilustrada na Figura 5.3, a restrição acima viola 44 cláusulas. Para essa restrição, uma cláusula é violada quando o lado esquerdo da implicação é verdadeiro enquanto o lado direito é falso. Esse é o caso quando $IN_i = V$, $EMPTY_i = F$ e $PARENT_{ijk} = F$ para algum valor de i .

Uma análise da Figura 5.3 revela que existem quatro valores de i para os quais $IN_i = V$ e $EMPTY_i = F$. Uma vez que as restrições WTA garantem que exista apenas um par (j, k) para o qual $PARENT_{ijk}$ é verdadeiro para um dado i , o número de cláusulas insatisfeitas na restrição 1(b) é igual a:

$$4 \times (2d - 1) = 4 \times (2 \times 6 - 1) = 44 \text{ cláusulas}$$

Essas violações de cláusulas são frutos da simplificação do existencial implícito encontrado na restrição 1(b): deseja-se, inicialmente, que para cada valor de i exista *apenas um* par (j, k) para o qual a fórmula seja satisfeita. No entanto, transformar esse existencial em quantificadores *para todo* acarreta uma substancial simplificação na função de energia. Adicionalmente, as restrições WTA ainda garantiriam a validade do *ou-exclusivo*.

De fato, na solução apresentada na Figura 5.3, cada linha da área de prova que não é uma cláusula vazia é pai de no máximo outra única linha da área de prova. No entanto, de acordo com a restrição simplificada 1(b), a afirmação é feita *para todos* os possíveis valores do par (j, k) , ou seja para todas as $2d^2$ cláusulas geradas por essa restrição. Se, devido às WTAs, apenas uma cláusula pode satisfazer a essa restrição para cada valor de i , então todas as outras cláusulas a violam para este mesmo valor de i .

A solução da Figura 5.4, obtida por SATyrus2, viola apenas 11 cláusulas dentro da restrição 1(b). Uma análise semelhante pode ser feita levando-se em conta as

demais restrições presentes na modelagem ARQ-PROP II. A Tabela 5.2 compara os custos entre as duas soluções aqui consideradas para todas as outras restrições da modelagem (apenas os custos diferentes de zero são exibidos):

Restrição	Cláusulas violadas na solução esperada	Cláusulas violadas na solução obtida
1(<i>b</i>)	44	11
2(<i>a</i>)	15	5
4(<i>a</i>)	118	120
4(<i>b</i>)	10	10

Tabela 5.2: Custos de ambas as soluções (esperada e obtida) para cada uma das restrições da modelagem anotada da ARQ-PROP II.

A partir da Tabela 5.2, pode-se calcular os custos de ambas as soluções. Como todas as restrições que violam cláusulas estão no nível de penalidade mais baixo, o valor numérico da penalidade associada a estas restrições é igual a 1. Portanto, a fração do custo total referente a cada restrição é exatamente igual ao número de cláusulas violadas por cada uma delas. Para a solução esperada, o custo é de $44 + 15 + 118 + 10 = 187$. Por outro lado, para a solução obtida pelo SATyrus2, o custo é igual a $11 + 5 + 120 + 10 = 136$. Portanto, a solução ilustrada na Figura 5.3, que esperava-se ser ótima, não é, de fato, a solução de custo mínimo para esta instância do problema.

Acreditava-se que as restrições WTA seriam suficientes para garantir a obtenção da solução ótima para o problema da ARQ-PROP II, ainda que a existência de cláusulas violadas fosse esperada devido à natureza das restrições do nível de penalidades 1. De fato, as WTAs conseguiram garantir a validade do *ou-exclusivo* para essas restrições; contudo, o ótimo global não pôde ser alcançado por meio desse método. Isso sugere a necessidade de uma correção na modelagem proposta em [28], com o intuito de reparar as imperfeições que ocasionaram a obtenção de uma configuração inconsistente e menos custosa do que a solução que se esperava ser ótima. Uma sugestão de melhoria para a modelagem da ARQ-PROP II está descrita no próximo capítulo.

5.3 Avaliação qualitativa

Ainda que o método de modelagem da ARQ-PROP II tenha se revelado insuficiente, o compilador SATyrus2 se mostrou apto a tratar problemas de grande porte. O resolvedor Bonmin, em conjunto com a plataforma AMPL, também se revelou uma boa alternativa para o processo de minimização da função de energia, obtendo para a modelagem da ARQ-PROP II, sem o auxílio de parâmetros e configurações adicionais, uma solução mais barata do que a menor solução conhecida até a realização

deste trabalho.

Outros testes de grande porte não foram levados a cabo em virtude da ausência, até o momento, de problemas maiores modelados em linguagem SATish. No entanto, a própria ARQ-PROP II pode ser remodelada e posteriormente reavaliada sob a plataforma SATyrus2. Esta sugestão faz parte da seção de trabalhos futuros do Capítulo 6, onde se encontram as considerações finais a respeito deste trabalho.

Capítulo 6

Conclusões

Este capítulo traz o encerramento de todo o trabalho apresentado até aqui. Após um resumo dos tópicos percorridos nesta tese e uma discussão dos resultados alcançados, são listados alguns exemplos de trabalhos futuros que podem ser tomados a partir da pesquisa realizada aqui.

6.1 Sumário

Os objetivos apontados no início deste trabalho foram três: **(i)** a expansão e aprimoramento da linguagem de modelagem reconhecida pela primeira versão da plataforma SATyrus; **(ii)** o desenvolvimento de um compilador (SATyrus2) capaz de traduzir essa linguagem em uma função de energia a ser minimizada por softwares de otimização; **(iii)** testes da plataforma SATyrus2 através da compilação da ARQ-PROP II, uma modelagem que contém um elevado número de variáveis e restrições.

Os objetivos quanto à linguagem e compilador foram alcançados. SATish apresenta melhorias em relação a sua antecessora, melhorias estas apresentadas na Seção 4.1.3. O compilador SATyrus2 foi desenvolvido com sucesso e se mostrou capaz de reconhecer e compilar especificações escritas em linguagem SATish, dentre elas modelagens mais extensas como a ARQ-PROP II. Também foi constatada a eficácia da plataforma AMPL + Bonmin no papel de resolvedores dos problemas de programação linear inteira criados pelo SATyrus2.

Os testes com ARQ-PROP II, no entanto, se revelaram problemáticos. Ainda que o SATyrus2 fosse capaz de compilar toda a modelagem da ARQ-PROP II em tempo hábil, os resultados eram inconclusivos. Não se sabia se a discrepância entre o resultado esperado e o resultado obtido pelo SATyrus2 era um problema no compilador ou na própria modelagem da ARQ-PROP II. Por fim, foi descoberta a fonte dos problemas: o método de eliminar existenciais e substituí-los por restrições WTA não garante necessariamente o menor custo para as soluções corretas. Ainda que este método construía soluções viáveis (isto é, que não violam WTAs), as res-

trições de existencial implícito são responsáveis por resíduos na função de energia que podem impedir a obtenção da solução esperada por parte do SATyrus2 e seus resolvedores.

A Seção 6.2 apresenta uma sugestão modelagem alternativa para a ARQ-PROP II, dentre outros trabalhos futuros.

6.2 Trabalhos futuros

Durantes os processos de desenvolvimento e testes do SATyrus2, notou-se que o compilador, ainda que funcional, pode ser melhorado em determinados pontos. Melhorias que dizem respeito à praticidade e à facilidade de uso do compilador, bem como à qualidade das respostas por ele geradas. Além disso, o SATyrus2 não representa, de modo algum, a única maneira com a qual se pode atacar o problema de modelagem e solução de problemas de otimização: pontos de partida inteiramente diferentes podem ser tomados.

Alguns dos trabalhos que podem se basear na pesquisa realizada aqui são:

- **Estudo de modelagens em linguagem AMPL:**

A linguagem AMPL oferece estruturas de dados e recursos que, a exemplo dos recursos existentes na linguagem SATish, podem ser utilizados na tarefa de modelar um problema de otimização. Considere, como exemplo, o código da Listagem 6.1, escrito em linguagem AMPL, equivalente à modelagem do TSP descrita na Seção 2.2.4:

```

param N integer ;

param cost > 0 ;
param int1 > 0 ;
param wta > 0 ;

param DIST { 1..N, 1..N } >= 0 ;

var POS { 1..N, 1..N } binary ;

minimize energy_function :
    int1 * ( 1 - POS [ 1 , 1 ] ) +
    int1 * ( 1 - POS [ 5 , 5 ] ) +

    int1 * sum { i in 1..N, j in 1..N }

```

$$\begin{aligned}
& (1 - \text{POS}[i, j]) + \\
& \text{wta} * \text{sum}\{i \text{ in } 1..N, j \text{ in } 1..N, k \text{ in } 1..N: i \triangleleft k\} \\
& (\text{POS}[i, j] * \text{POS}[k, j]) + \\
& \text{wta} * \text{sum}\{i \text{ in } 1..N, j \text{ in } 1..N, l \text{ in } 1..N: j \triangleleft l\} \\
& (\text{POS}[i, j] * \text{POS}[i, l]) + \\
& \text{cost} * \text{sum}\{i \text{ in } 1..N, j \text{ in } 2..N, k \text{ in } 1..N: i \triangleleft k\} \\
& (\text{DIST}[i, k] * (\text{POS}[i, j] * \text{POS}[k, j-1])) + \\
& \text{cost} * \text{sum}\{i \text{ in } 1..N, j \text{ in } 1..N-1, k \text{ in } 1..N: i \triangleleft k\} \\
& (\text{DIST}[i, k] * (\text{POS}[i, j] * \text{POS}[k, j+1]));
\end{aligned}$$

Listagem 6.1: TSP modelado em linguagem AMPL.

O código da Listagem 6.1 é uma aplicação manual do mapeamento apresentado na Seção 2.2.1 à forma clausal do TSP apresentada na página 10. Na modelagem acima, as estruturas **DIST** e **POS** têm papel idêntico às estruturas de mesmo nome presentes nas modelagens do TSP apresentadas até aqui. As penalidades são representadas pelos parâmetros *cost*, *int1* e *wta*, que atuam como constantes multiplicativas dentro da função de energia (denominada, no código acima, por “energy_function”).

Em AMPL, os valores dos parâmetros podem ser estabelecidos com o auxílio de um arquivo secundário, utilizado exclusivamente como entrada de dados. A Figura 6.1 ilustra o formato desse arquivo. Um exemplo de arquivo de controle AMPL que pode ser utilizado para se executar a modelagem AMPL da Listagem 6.1 está ilustrado na Figura 6.2.

```

data;

param N := 5;

param cost := 9;
param int1 := 720;
param wta := 18720;

param DIST :
    1  2  3  4  5 :=
    1  0  4  4  9  0
    2  4  0  9  4  4
    3  4  9  0  4  4
    4  9  4  4  0  9
    5  0  4  4  9  0 ;

```

Figura 6.1: Dados de entrada para a modelagem da Listagem 6.1.

```
model tsp.ampl;
data tsp.dat;
option solver bonmin;
options bonmin_options "bonmin.num_resolve_at_root 2";
options bonmin_options "bonmin.num_resolve_at_node 4";
solve;
display {i in 1.._nvars} (_varname[i], _var[i]);
```

Figura 6.2: Exemplo de arquivo de controle AMPL que resolve a Listagem 6.1.

O arquivo de controle ilustrado na Figura 6.2 contém duas instruções específicas direcionadas ao resolvidor Bonmin: a primeira, *num_resolve_at_root* = 2, faz com que o Bonmin escolha dois pontos de partida aleatórios para o processo de otimização. O ponto que fornecer uma valor de menor custo para a função de energia é então utilizado durante todo o restante do processo de busca por uma solução ótima. A segunda instrução, *num_resolve_at_node* = 4, faz com que Bonmin teste quatro soluções candidatas a cada passo do processo de minimização. A solução candidata que fornecer o menor valor da função de energia é a escolhida para o restante do procedimento.

A Figura 6.3 exibe a solução encontrada por AMPL quando se resolve o modelo descrito na Listagem 6.1. A solução corresponde, de fato, ao ótimo do problema.

A compilação de modelos na plataforma SATyrus2 contém um estágio de *enumeração* de cláusulas: todas as cláusulas resultantes de cada uma das restrições presentes no modelo de entrada são armazenadas em memória e posteriormente convertidas em expressões algébricas que compõem a função de energia. No entanto, restrições que possuam muitos índices podem gerar um número alto de cláusulas, todas presentes em memória simultaneamente. O arquivo de modelagem resultante, escrito em linguagem AMPL, pode ser extenso, devido ao alto número de expressões algébricas.

Esse problema pode ser contornado por meio de modelagens feitas diretamente em AMPL, uma vez que recursos da linguagem, como intervalos e *loops*, podem ser utilizados na elaboração de uma função de energia escrita em forma compacta, e não aberta, como são as funções de energia geradas pelo SATyrus2.

Modelagens semelhantes à Listagem 6.1, no entanto, trazem duas dificuldades: em primeiro lugar, pode ser incômodo trabalhar diretamente com expressões algébricas no lugar de fórmulas lógicas durante o processo de modelagem; em segundo lugar, esse tipo de modelagem exige que os valores das penalidades sejam calculados manualmente pelo usuário e fornecidos como parâmetros de entrada para a plataforma AMPL.

```

bonmin: Optimal
:   _varname[i] _var[i]   :=
1   'POS[1,1]'   1
2   'POS[1,2]'   0
3   'POS[1,3]'   0
4   'POS[1,4]'   0
5   'POS[1,5]'   0
6   'POS[2,1]'   0
7   'POS[2,2]'   1
8   'POS[2,3]'   0
9   'POS[2,4]'   0
10  'POS[2,5]'   0
11  'POS[3,1]'   0
12  'POS[3,2]'   0
13  'POS[3,3]'   0
14  'POS[3,4]'   1
15  'POS[3,5]'   0
16  'POS[4,1]'   0
17  'POS[4,2]'   0
18  'POS[4,3]'   1
19  'POS[4,4]'   0
20  'POS[4,5]'   0
21  'POS[5,1]'   0
22  'POS[5,2]'   0
23  'POS[5,3]'   0
24  'POS[5,4]'   0
25  'POS[5,5]'   1
;

```

Figura 6.3: Solução encontrada por AMPL para o modelo da Listagem 6.1.

- **Compilador de dois ou mais passos:**

O SATyrus2 é um compilador de um só passo, o que significa que o código fonte dos arquivos de modelagem é percorrido uma única vez durante todo o processo de compilação. Esse tipo de abordagem implica em ganho de desempenho; porém, traz restrições à linguagem que pode ser reconhecida pelo compilador. A reescrita de parte do código do SATyrus2 de modo a transformá-lo em um compilador de dois ou mais passos poderia trazer as seguintes vantagens:

- **Arquivos de modelagem com estrutura mais flexível:** os modelos escritos em linguagem SATish não mais precisariam seguir a ordem ‘declaração de estruturas, definição de restrições e atribuição de penalidades’. Essas seções poderiam ser rearranjadas e até intercaladas dentro dos arquivos de modelagem, a gosto do usuário.
- **Estrutura modular para o Parser:** o módulo Parser poderia ser dividido em diversos submódulos, cada um deles responsável por um estágio do processo de reconhecimento e validação de códigos fonte: análise léxica, análise sintática, análise de contexto etc.
- **Geração de Árvore Sintática Abstrata:** os modelos escritos em linguagem SATish poderiam ser convertidos em Árvores Sintáticas Abstratas (*Abstract Syntax Trees*, ou ASTs), representações internas das estruturas sintáticas dos códigos fontes submetidos a compilação. As ASTs podem ser utilizadas, por exemplo, na geração automática de modelos AMPL semelhantes ao código exibido na Listagem 6.1.

- **Especificação de arquivos de dados através da linha de comando:**

Ainda que suporte a especificação de arquivos de dados externos (veja a Seção 4.1.3), SATyrus2 demanda que os mesmos sejam referenciados no código fonte dos modelos escritos em SATish. A especificação desses arquivos pode ser movida para o módulo de Interface com o usuário; desse modo, arquivos de dados semelhantes àqueles utilizados por AMPL podem ser construídos pelo usuário e fornecidos ao compilador na forma de parâmetros, como a seguir:

```
$ satyrus --data tsp.dat --model tsp.sat -o tsp
```

Essa abordagem evita a edição desnecessária de código, permitindo que o mesmo arquivo de modelagem seja facilmente utilizado com diversas instâncias de dados diferentes, como a seguir:

```
$ satyrus --data tsp.dat1 --model tsp.sat -o tsp1
$ satyrus --data tsp.dat2 --model tsp.sat -o tsp2
$ satyrus --data tsp.dat3 --model tsp.sat -o tsp3
```

- **Estudo das opções de resolução do Bonmin:**

Bonmin utiliza o algoritmo de *Branch and Bound* para varrer o espaço de soluções candidatas da função de energia à procura de um mínimo global. Mesmo que Bonmin seja capaz de trabalhar com outros algoritmos, *Branch and Bound* é o único método recomendado pelos desenvolvedores da plataforma para a resolução de problemas de otimização não-convexa [26]. No entanto, Bonmin oferece ao usuário uma série de opções de resolução, tais como *num_resolve_at_root* e *num_resolve_at_node*, citadas no primeiro item desta seção. Os impactos da variação dessas opções durante o processo de resolução poderiam ser mensurados e estudados, dando origem a uma estratégia eficiente de obtenção de soluções próximas ao mínimo global para as equações de energia construídas pelo SATyrus2.

- **Estudo do funcionamento de outros resolvedores:**

Em todos os exemplos e testes descritos neste trabalho, o resolvedor utilizado para minimizar a função de energia foi Bonmin. Entretanto, este não é o único resolvedor que faz interface com AMPL e que pode ser utilizado diretamente através do SATyrus2. Exemplos de resolvedores capazes de resolver problemas de programação inteira não-convexa e que podem ser utilizados através de AMPL são Couenne [25] e FilMINT [24]. Tais resolvedores podem ser explorados, avaliados e comparados com Bonmin quanto à sua eficácia na resolução de problemas de programação matemática semelhantes àqueles gerados pelo SATyrus2.

- **Estudo mais aprofundado da plataforma Xpress:**

Durante a fase de testes com a ARQ-PROP II, a plataforma Xpress se comportou de maneira estranha quanto ao valor das variáveis binárias declaradas nos modelos construídos pelo SATyrus2. Algumas variáveis passavam por um processo de *relaxação*, no qual valores diferentes de 0 e 1 passam a ser atribuições válidas às variáveis binárias. Este é um método válido de otimização desde que a solução final seja inteiramente binária. No entanto, as soluções apresentadas por Xpress para a ARQ-PROP II continha variáveis para as quais foram atribuídos valores reais no intervalo $(0, 1)$, que representam coordenadas inválidas dentro do espaço de soluções da ARQ-PROP II. Pode-se estudar os motivos desse comportamento. Também pode-se pesquisar um método de utilização da

plataforma Xpress de modo a torná-la tão eficiente quanto o conjunto AMPL + Bonmin no contexto dos problemas criados pelo SATyrus2.

- **Aprimoramento na modelagem da ARQ-PROP II:**

A linguagem SATish foi desenvolvida sob demanda: foram nela embutidos apenas os recursos necessários às modelagens do TSP, do problema da Coloração de Grafos e da ARQ-PROP II. Contudo, como discutido no Capítulo 5, a modelagem da ARQ-PROP II não funcionou como esperado. O problema constatado foi a violação de cláusulas em restrições nas quais um existencial implícito foi convertido em uma série de conjunções. Essa abordagem simplifica a função de energia; no entanto, também insere imprecisões que impedem o correto funcionamento do sistema de raciocínio lógico que a ARQ-PROP II implementa.

Uma das possíveis maneiras de melhorar a modelagem da ARQ-PROP II é expandir a linguagem SATish com a inclusão de um quantificador lógico que represente o *ou-exclusivo* que se deseja obter para determinados conjuntos de cláusulas. O novo quantificador, que deve ser implementado no código fonte do SATyrus2, garantiria a validade de apenas uma cláusula dentro da definição de uma restrição qualquer. A linguagem poderia ser expandida como representado na Listagem 6.2.

```
forall {i} where i in (1,d);
    exists_one {j} where j in (1,d):
        (...)
```

Listagem 6.2: Sugestão de sintaxe para o quantificador de ou-exclusivo.

Ainda que custosa em termos de função de energia, essa abordagem pode ser uma saída para o problema da ARQ-PROP II apresentado no capítulo anterior. O usuário do compilador deveria ficar ciente dos custos desse procedimento.

Referências Bibliográficas

- [1] *Python Programming Language*, . Disponível em: <<http://www.python.org/>>.
- [2] *AMPL Modeling Language for Mathematical Programming*, . Disponível em: <<http://www.ampl.com/>>.
- [3] *FICOTM Xpress Optimization Suite 7*. Disponível em: <<http://www.fico.com/en/Products/DMTools/Pages/FICO-Xpress-Optimization-Suite.aspx>>.
- [4] *FICO Xpress-Mosel*. Disponível em: <<http://www.fico.com/en/Products/DMTools/xpress-overview/Pages/Xpress-Mosel.aspx>>.
- [5] ROBERT FOURER, D. M. G., KERNIGHAN, B. W. “A Modeling Language for Mathematical Programming”, *Management Science*, v. 36, pp. 519–554, 1990.
- [6] ESPINOZA, M. M. M. M. *Compilando Resolução de Problemas para Minimização de Energia*. Tese de Mestrado, Universidade Federal do Rio de Janeiro, COPPE, 2006.
- [7] *Bazaar*, . Disponível em: <<http://bazaar.canonical.com/>>.
- [8] *Python Lex-Yacc*. Disponível em: <<http://www.dabeaz.com/ply/>>.
- [9] *Computer Architecture and Microelectronics Laboratory (PESC/COPPE/UFRJ)*. Disponível em: <<http://www.lam.ufrj.br/>>.
- [10] *Bazaar Documentation*, . Disponível em: <<http://doc.bazaar.canonical.com/>>.
- [11] *Sympy*. Disponível em: <<http://sympy.org/>>.
- [12] *Built-in Functions - Python documentation*, . Disponível em: <<http://docs.python.org/library/functions.html>>.
- [13] *Solvers that Work with AMPL*, . Disponível em: <<http://www.ampl.com/solvers.html>>.

- [14] *Hooking Your Solver to AMPL*, . Disponível em: <<http://www.ampl.com/REFS/HOOKING/>>.
- [15] COOK, S. A. “The complexity of theorem-proving procedures”. In: *STOC ’71: Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158, New York, NY, USA, 1971. ACM.
- [16] *Bonmin Home Page*. Disponível em: <<http://www.coin-or.org/Bonmin/>>.
- [17] OVERVIEW, M. A., COLOMBANI, Y., HEIPCKE, S. “Mosel: An Overview Dash Optimization Whitepaper Contents”. , 2007.
- [18] WILSON, P. *Design Recipes for FPGAs*. 1ª ed. Burlington, MA, UK, Newnes, 2007.
- [19] LIMA, P. M. V., MORVELI-ESPINOZA, M. M., FRANÇA, F. M. G. “Logic as Energy: A SAT-Based Approach”. In: *BVAI*, pp. 458–467, 2007.
- [20] PINKAS, G. *Logical Inference in Symmetric Neural Networks*. Tese de Doutorado, Sever Institute of Technology, 1992.
- [21] LIMA, P. M. V., MORVELI-ESPINOZA, M. M., PEREIRA, G. C., et al. “SATyrus: A SAT-based Neuro-Symbolic Architecture for Constraint Processing”, *Hybrid Intelligent Systems, International Conference on*, v. 0, pp. 137–142, 2005.
- [22] PEREIRA, G. *Mapeamento e Combinação de Problemas NP-Difíceis, através de Restrições Pseudo-Booleanas para Redes Neurais Artificiais*. Tese de Mestrado, Universidade Federal do Rio de Janeiro, COPPE, 2006.
- [23] GEMAN, S., GEMAN, D. “Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images”, pp. 564–584, 1987.
- [24] ABHISHEK, K., LEYFFER, S., LINDEROTH, J. T. *FilMINT: An outer-approximation-based solver for nonlinear mixed integer programs*. Relatório Técnico ANL/MCS-P1374-0906, Argonne National Laboratory, Mathematics and Computer Science Division, 2006.
- [25] BELOTTI, P., LEE, J., LIBERTI, L., et al. “Branching and bounds tightening techniques for non-convex MINLP”, *Optimization Methods Software*, v. 24, n. 4-5, pp. 597–634, 2009. ISSN: 1055-6788.
- [26] BONAMI, P., LEE, J. *BONMIN Users’ Manual*, 2007. Disponível em: <https://projects.coin-or.org/Bonmin/browser/stable/0.1/Bonmin/doc/BONMIN_UsersManual.pdf?format=raw>.

- [27] LIMA, P. M. V., PEREIRA, G. C., MORVELI-ESPINOZA, M. M., et al. “Mapping and Combining Combinatorial Problems into Energy Landscapes via Pseudo-Boolean Constraints”. In: *BVAI*, pp. 308–317, 2005.
- [28] LIMA, P. M. V. “A Goal-Driven Neural Propositional Interpreter”, *Int. J. Neural Syst.*, v. 11, n. 3, pp. 311–322, 2001.
- [29] LIMA, P. M. V. “A Neural Propositional Reasoner that is Goal-Driven and Works without Pre-Compiled Knowledge”. In: *SBRN '00: Proceedings of the VI Brazilian Symposium on Neural Networks (SBRN'00)*, p. 261, Washington, DC, USA, 2000. IEEE Computer Society. ISBN: 0-7695-0856-1.
- [30] LEVINE, J. R., MASON, T., BROWN, D. *lex & yacc (2nd ed.)*. Sebastopol, CA, USA, O'Reilly & Associates, Inc., 1992. ISBN: 1-56592-000-7.
- [31] FORSBERG, M., RANTA, A. “BNF converter”. In: *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pp. 94–95, New York, NY, USA, 2004. ACM. ISBN: 1-58113-850-4.
- [32] FORSBERG, M., RANTA, A. *The Labelled BNF Grammar Formalism*. Relatório técnico, Chalmers University of Technology and the University of Gothenburg, 2005.

Apêndice A

SATish - Language Specification

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure of grammar

Identifiers

Identifiers $\langle Ident \rangle$ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_` `'`, reserved words excluded.

Literals

String literals $\langle String \rangle$ have the form `"x"`, where x is any sequence of any characters except `"` unless preceded by `\`.

Integer literals $\langle Int \rangle$ are nonempty sequences of digits.

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in grammar are the following:

and exists forall
 from in intgroup
 level not optgroup
 or penalties where

The symbols used in grammar are the following:

; = (
) , ==
 != []
 : { }
 -> <- <->
 + - *
 /

Comments

Single-line comments begin with `//`.

Multiple-line comments are enclosed with `/*` and `*/`.

The syntactic structure of grammar

Non-terminals are enclosed between \langle and \rangle . The symbols $::=$ (production), $|$ (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

$$\langle \text{Program} \rangle ::= \langle \text{ListDefinition} \rangle \langle \text{ListConstraint} \rangle \langle \text{Penalties} \rangle$$

$$\langle \text{ListDefinition} \rangle ::= \epsilon$$

$$| \quad \langle \text{Definition} \rangle ; \langle \text{ListDefinition} \rangle$$

$$\langle \text{Definition} \rangle ::= \langle \text{Ident} \rangle = \langle \text{Expr} \rangle$$

$$| \quad \langle \text{Ident} \rangle (\langle \text{Dimensions} \rangle)$$

$$| \quad \langle \text{Ident} \rangle = \langle \text{StructBody} \rangle$$

$$| \quad \langle \text{Ident} \rangle \text{ from } \langle \text{String} \rangle$$

$$\langle \text{Dimensions} \rangle ::= \langle \text{ListExpr} \rangle$$

$$| \quad \langle \text{Domain} \rangle$$

$$\langle \text{Domain} \rangle ::= \langle \text{ListInterval} \rangle$$

$$| \quad \langle \text{ListInterval} \rangle \text{ and } \langle \text{ListAssertion} \rangle$$

$$\begin{aligned}
\langle \text{ListInterval} \rangle &::= \langle \text{Interval} \rangle \\
&\quad | \quad \langle \text{Interval} \rangle , \langle \text{ListInterval} \rangle \\
\langle \text{Interval} \rangle &::= \langle \text{Ident} \rangle \text{ in } (\langle \text{Expr} \rangle , \langle \text{Expr} \rangle) \\
\langle \text{ListAssertion} \rangle &::= \langle \text{Assertion} \rangle \\
&\quad | \quad \langle \text{Assertion} \rangle , \langle \text{ListAssertion} \rangle \\
\langle \text{Assertion} \rangle &::= \langle \text{Ident} \rangle == \langle \text{Iexpr} \rangle \\
&\quad | \quad \langle \text{Ident} \rangle != \langle \text{Iexpr} \rangle \\
\langle \text{StructBody} \rangle &::= [\langle \text{ListMapping} \rangle] \\
\langle \text{ListMapping} \rangle &::= \langle \text{Mapping} \rangle \\
&\quad | \quad \langle \text{Mapping} \rangle ; \langle \text{ListMapping} \rangle \\
\langle \text{Mapping} \rangle &::= \langle \text{Key} \rangle : \langle \text{Integer} \rangle \\
\langle \text{Key} \rangle &::= \langle \text{ListInteger} \rangle \\
&\quad | \quad (\langle \text{ListInteger} \rangle) \\
\langle \text{ListInteger} \rangle &::= \langle \text{Integer} \rangle \\
&\quad | \quad \langle \text{Integer} \rangle , \langle \text{ListInteger} \rangle \\
\langle \text{ListConstraint} \rangle &::= \epsilon \\
&\quad | \quad \langle \text{Constraint} \rangle ; \langle \text{ListConstraint} \rangle \\
\langle \text{Constraint} \rangle &::= \text{intgroup } \langle \text{Ident} \rangle : \langle \text{ConstraintRHS} \rangle \\
&\quad | \quad \text{optgroup } \langle \text{Ident} \rangle : \langle \text{ConstraintRHS} \rangle \\
\langle \text{ConstraintRHS} \rangle &::= \langle \text{Quantifiers} \rangle \langle \text{Formulae} \rangle \\
\langle \text{Quantifiers} \rangle &::= \epsilon \\
&\quad | \quad \langle \text{ListQuantifier} \rangle : \\
\langle \text{ListQuantifier} \rangle &::= \langle \text{Quantifier} \rangle \\
&\quad | \quad \langle \text{Quantifier} \rangle ; \langle \text{ListQuantifier} \rangle \\
\langle \text{Quantifier} \rangle &::= \text{forall } \{ \langle \text{ListIdent} \rangle \} \text{ where } \langle \text{Domain} \rangle \\
&\quad | \quad \text{exists } \{ \langle \text{ListIdent} \rangle \} \text{ where } \langle \text{Domain} \rangle \\
\langle \text{ListIdent} \rangle &::= \langle \text{Ident} \rangle \\
&\quad | \quad \langle \text{Ident} \rangle , \langle \text{ListIdent} \rangle \\
\langle \text{Formulae} \rangle &::= \langle \text{WWF} \rangle \\
&\quad | \quad \langle \text{Integer} \rangle (\langle \text{WWF} \rangle) \\
&\quad | \quad \langle \text{Atom} \rangle (\langle \text{WWF} \rangle)
\end{aligned}$$

$$\begin{aligned}
\langle \text{WWF} \rangle &::= \langle \text{WWF} \rangle \text{ or } \langle \text{WWF2} \rangle \\
&| \quad \langle \text{WWF} \rangle \text{ and } \langle \text{WWF2} \rangle \\
&| \quad \langle \text{WWF2} \rangle \\
\langle \text{WWF2} \rangle &::= \langle \text{WWF2} \rangle \rightarrow \langle \text{WWF3} \rangle \\
&| \quad \langle \text{WWF2} \rangle <- \langle \text{WWF3} \rangle \\
&| \quad \langle \text{WWF2} \rangle <-> \langle \text{WWF3} \rangle \\
&| \quad \langle \text{WWF3} \rangle \\
\langle \text{WWF3} \rangle &::= \text{not } \langle \text{WWF4} \rangle \\
&| \quad \langle \text{WWF4} \rangle \\
\langle \text{WWF4} \rangle &::= \langle \text{Atom} \rangle \\
&| \quad (\langle \text{WWF} \rangle) \\
\langle \text{Atom} \rangle &::= \langle \text{Ident} \rangle \\
&| \quad \langle \text{Ident} \rangle \langle \text{ListIndex} \rangle \\
\langle \text{ListIndex} \rangle &::= \langle \text{Index} \rangle \\
&| \quad \langle \text{Index} \rangle \langle \text{ListIndex} \rangle \\
\langle \text{Index} \rangle &::= [\langle \text{Iexpr} \rangle] \\
\langle \text{ListExpr} \rangle &::= \langle \text{Expr} \rangle \\
&| \quad \langle \text{Expr} \rangle , \langle \text{ListExpr} \rangle \\
\langle \text{Expr} \rangle &::= \langle \text{Expr} \rangle + \langle \text{Expr2} \rangle \\
&| \quad \langle \text{Expr} \rangle - \langle \text{Expr2} \rangle \\
&| \quad \langle \text{Expr2} \rangle \\
\langle \text{Expr2} \rangle &::= \langle \text{Expr2} \rangle * \langle \text{Expr3} \rangle \\
&| \quad \langle \text{Expr2} \rangle / \langle \text{Expr3} \rangle \\
&| \quad \langle \text{Expr3} \rangle \\
\langle \text{Expr3} \rangle &::= - \langle \text{Expr4} \rangle \\
&| \quad \langle \text{Expr4} \rangle \\
\langle \text{Expr4} \rangle &::= \langle \text{Integer} \rangle \\
&| \quad \langle \text{Ident} \rangle \\
&| \quad (\langle \text{Expr} \rangle) \\
\langle \text{Iexpr} \rangle &::= \langle \text{Ident} \rangle \\
&| \quad \langle \text{Integer} \rangle \\
\langle \text{Penalties} \rangle &::= \text{penalties} : \langle \text{ListPenalty} \rangle \\
\langle \text{ListPenalty} \rangle &::= \langle \text{Penalty} \rangle ; \\
&| \quad \langle \text{Penalty} \rangle ; \langle \text{ListPenalty} \rangle
\end{aligned}$$

$\langle \textit{Penalty} \rangle ::= \langle \textit{Ident} \rangle \textit{level} \langle \textit{Integer} \rangle$

Apêndice B

Manual do compilador

Este apêndice é um guia sobre SATyrus2. A Seção B.1 lista os passos necessários para a obtenção do código fonte do compilador; a Seção B.2 traz instruções de instalação do SATyrus2; a Seção B.3 detalha as opções oferecidas pelo compilador; por fim, a Seção B.5 traz uma lista de todos os possíveis erros de compilação retornados pelo SATyrus2.

B.1 Obtendo o código fonte do compilador

Todo o desenvolvimento do código fonte do SATyrus2 foi mantido sob controle de versão. O Bazaar [7] foi o sistema de versionamento escolhido. Para adquirir uma cópia do código fonte do SATyrus2, bem como toda a história de sua implementação, é necessário possuir autorização para acesso remoto ao Laboratório de Arquitetura e Microeletrônica da Coppe/UFRJ [9]. De posse de um nome de usuário válido no laboratório e de uma instalação operante do Bazaar, o código fonte pode ser copiado através de um terminal de comandos da seguinte maneira:

```
$ bazaar branch http://www.lam.ufrj.br/~bruno/bazaar/satyrus
```

Serão requeridos o nome de usuário e a correspondente senha de acesso. Após a conclusão da cópia do código fonte do SATyrus2, a história do desenvolvimento pode ser visualizada através dos comandos:

```
$ cd satyrus
$ bazaar log --include-merges
```

Mais detalhes a respeito das opções de visualização oferecidas pelo Bazaar podem ser encontrados em [10].

B.2 Instalação

O SATyrus2 possui duas dependências: Python (versão 2.4, 2.5 ou 2.6) e a biblioteca de *parsing* PLY [8]. Satisfeitas as dependências, o SATyrus2 pode ser instalado através do seguinte comando, que deve ser executado a partir do diretório no qual se encontra o código fonte do compilador:

```
$ python setup.py install
```

O comando acima instala o executável **satyrus**, que implementa o módulo de Interface com o usuário (ver Seção 4.3). O comando também instala todos os arquivos necessários ao funcionamento do compilador.

B.3 Opções de compilação

As opções de compilação presentes no SATyrus2 não são de uso obrigatório. Em uma compilação padrão, basta que o usuário forneça o nome do arquivo de modelagem que se deseja compilar, como a seguir:

```
$ satyrus model.sat
```

No entanto, o SATyrus2 oferece opções que modificam o fundamento padrão do compilador. Essas opções podem ser visualizadas num terminal de comando através da opção **--help**, passada ao executável **satyrus**, como ilustra a Figura B.1:

```
$ satyrus --help
Usage: satyrus [options] file

Options:
  --version            show program's version number and exit
  -h, --help           show this help message and exit
  -u num, --upper-bound=num
                        opt penalties upper bound (this can be any real
                        number greater or equal to 1)
  -s, --simplify       simplify the energy function doing things like
                        x + x => 2*x. Warning: this can be *really* slow
  -a, --ampl           generate ampl output (default)
  -x, --xpress         generate xpress (mosel) output
  -o PREFIX           output files prefix. Default is 'out' (for
                        instance, ampl output files would be 'out.mod'
                        and 'out.in').

Ampl Options:
  --solver=SOLVER      solver that will be used to solve the energy
                        function. Default is 'bonmin'.
```

Figura B.1: Lista de opções disponíveis no SATyrus2

A opção **satyrus** exibe uma lista de opções de compilação disponíveis, assim como breves explicações a respeito de cada uma delas. Estas opções são:

- **--version**: exibe a versão corrente do compilador;
- **-h, --help**: imprime a mensagem ilustrada na Figura B.1;
- **-u num, --upper-bound=num**: limite superior das penalidades correspondentes às restrições de integridade. Útil em modelagens como as do TSP.
- **-s, --simplify**: ‘simplifica’ a função de energia. Por padrão, a função de energia é uma transcrição direta da fórmula gerada pelo Parser durante o processo de compilação, sem que nenhuma outra modificação seja realizada na função obtida. No entanto, caso esta opção seja especificada pelo usuário, a função de energia será simplificada por meio da realização de operações algébricas (desde que estas sejam possíveis). A Tabela B.1 ilustra esse procedimento.

Exemplo de <i>LMS</i>	$(a \vee b \vee a) \wedge (\neg a \vee \neg b)$
Função de energia padrão	$f(a, b) = (a + b + a) * ((1 - a) + (1 - b))$
Função de energia simplificada	$f(a, b) = (2 * a + b) * (2 - a - b)$

Tabela B.1: Exemplo de simplificação de função de energia

Observação: esta opção exige a instalação da biblioteca Sympy [11].

- **-a, --ampl**: gera modelos AMPL.
- **-x, --xpress**: gera modelos Xpress (Mosel).
- **-o PREFIXO**: concatena extensões apropriadas a PREFIXO para formar o nome dos arquivos de saída. As extensões utilizadas são *.in*, para arquivos de controle AMPL; *.mod*, para modelos AMPL; e *.mos*, para modelos Mosel.
- **--solver=SOLVER**: solver a ser utilizado em modelos AMPL. Caso esta opção seja omitida, Bonmin é o solver padrão escolhido pelo compilador.

A Seção B.4 traz alguns exemplos de utilização do SATyrus2 e suas opções de compilação.

B.4 Exemplos

O código fonte do SATyrus2 possui um diretório denominado **test**, onde encontram-se modelos que foram utilizados para testar o funcionamento do compilador durante o processo de desenvolvimento. Este diretório contém, entre outros, a modelagem do TSP discutida na Seção 3.3. Exemplos de uso do SATyrus2 em um terminal de comandos UNIX:

- gerando um modelo AMPL:

```
$ satyrus --ampl -o tsp test/tsp.sat
```

- gerando um modelo AMPL (forma simplificada):

```
$ satyrus -o tsp test/tsp.sat
```

- executando um modelo AMPL gerado pelo SATyrus2:

```
$ ampl < tsp.in
```

- gerando um modelo Xpress:

```
$ satyrus --xpress -o tsp test/tsp.sat
```

- executando um modelo Xpress gerado pelo SATyrus2:

```
$ mosel -c "exec tsp"
```

B.5 Erros de compilação

Possíveis erros de modelagem cometidos pelo usuário do compilador são imediatamente reportados pelo SATyrus2. Quando um erro é encontrado em uma modelagem qualquer fornecida pelo usuário, a compilação pára e uma mensagem de erro exibida no terminal de comando informa o usuário a respeito do erro cometido. Os erros que podem ser reportados pelo SATyrus2 são:

- **“illegal character ‘c’”**: um caractere inválido foi encontrado no arquivo fonte. Caracteres inválidos são caracteres que não fazem parte da linguagem SATish, definida no Apêndice A.
- **“unexpected end of file”**: o arquivo fonte contém um modelo incompleto. Exemplos de modelos incompletos são modelos que não possuem especificação de penalidades ou modelos que não possuem quaisquer restrições de integridade ou otimalidade.
- **“syntax error in input file”**: o arquivo fonte contém um modelo sintaticamente incorreto. Erros sintáticos são inadequações às regras de formação da linguagem SATish definidas em A.
- **“In order to run SATyrus, you must install X”**: a dependência *X* não foi satisfeita e, portanto, o compilador não pôde ser executado.

- **“neuron value must be a non-negative integer”**: ao menos uma valor negativo foi atribuído a um neurônio. Exemplo:

```
M(1,1);
M = { 1,1: -4 };
```

- **“index value must be a positive integer”**: um valor não positivo foi utilizado como índice. A indexação das estruturas neurais válidas no contexto da linguagem SATish admite apenas índices maiores ou iguais a 1.
- **“index out of bounds”**: pelo menos um índice que se encontra fora dos limites dimensionais definidos para uma certa estrutura foi utilizado. Exemplo:

```
M(4,4);
intgroup int0: M[1][8];
```

- **“can’t open file *X* for reading”**: erro de leitura no arquivo *X*. A linguagem SATish permite que o conteúdo de estruturas neurais seja importado a partir de arquivos externos ao modelo sendo compilado. Caso não seja possível ler estes arquivos (devido a inexistência dos mesmos, permissões insuficientes etc), este erro é reportado ao usuário.
- **“optimality constraints can’t be WTAs”**: uma restrição de otimalidade foi definida pelo usuário como uma WTA.
- **“int variable is unsubscriptable”**: uma constante foi utilizada como uma estrutura, como no exemplo abaixo:

```
N = 4;
intgroup int0: N[1];
```

- **“wrong number of dimensions for struct *X*”**: uma estrutura de dimensão *N* foi utilizada com menos ou mais do que *N* índices. Exemplo:

```
M(2,2);
intgroup int0: N[1][1][1];
```

- “**struct needs a subscript**”: pelo menos uma estrutura foi utilizada sem indexação, como no exemplo abaixo:

```
M(2,2);
N(2,2);
intgroup int0: M and N;
```

- “**constraint X is already in level Y**”: ao menos uma restrição foi definida em dois ou mais índices. Exemplo:

```
intgroup int1: A[1];
(...)
penalties:
    int1: level 1;
    int1: level 2;
```

- “**levels must be non-negative intergers**”: pelo menos um nível de penalidades foi associado a um peso negativo. Exemplo:

```
penalties:
    int1: level -1;
```

- “**invalid index: X**”: durante a declaração de uma dada estrutura, ao menos uma lista de índices possui tamanho maior do que o número de dimensões da mesma estrutura. Exemplo:

```
M(2,2);
M = { 1,1: 0;
```

```
1,2: 1;  
2,1: 1  
2,2,2: 0 };
```

- “**indexes must be integers**”: ao menos um índice que não é um número inteiro foi utilizado. Exemplo:

```
ingroup int0: M[1.2][3];
```

- “**X: unbound variable**”: a variável X se encontra fora dos limites estabelecidos para uma dada estrutura. Exemplo:

```
M(1,1);  
ingroup int1:  
    M[1][2] and M[1][3];
```

- “**name X is not defined**”: um identificador indefinido foi utilizado. No exemplo abaixo, o identificador c está indefinido, pois não foi declarado em nenhuma região do código:

```
M(1,1+c);  
ingroup int1: M[1][1];  
penalties:  
    int1: level 1;
```

- “**X should be of type Y**”: o identificador X foi utilizado em um contexto no qual se esperava uma expressão de tipo diferente. Exemplo:

```
M(1,1);  
ingroup int1: M[1][1];  
penalties:  
    M: level 1;
```

No exemplo acima, o identificador M , que representa uma estrutura, foi utilizado em um contexto no qual se esperava um identificador de penalidades.

Quaisquer outros erros exibidos pelo compilador constituem erros não esperados e, portanto, *bugs*. Estes devem ser reportados ao responsável pelo compilador (veja o arquivo README, distribuído junto com o código fonte do SATyrus2).

Apêndice C

Código fonte da ARQ-PROP II em linguagem SATish

```
n=6;
IN(n);
PROOF(n,n,2);
CB(n);
RES(n);
EMPTY(n);
ORIG(n);
CLCOMP(n,n,2);
CBMAP(n,n);
PARENT(n,n,2);
CANCEL(n,n);

PROOF  from "proof.txt";
CB      from "cb.txt";
RES     from "res.txt";
EMPTY  from "empty.txt";
ORIG    from "orig.txt";
CLCOMP from "clcomp.txt";
CBMAP  from "cbmap.txt";
PARENT from "parent.txt";
CANCEL from "cancel.txt";

CBMAP = [6,4: 1];
PARENT = [6,4,1: 1];
```



```

intgroup int1:
  forall{i} where i in (1,n):
    IN[i] -> CB[i] or RES[i];

intgroup int0:
  forall{i,j,k} where i in (1,n), j in (1,n), k in (1,2):
    IN[i] and not EMPTY[i] -> PARENT[i][j][k];

intgroup int1:
  forall{i,j,k} where i in (1,n), j in (1,n), k in (1,2):
    IN[i] and EMPTY[i] -> not PARENT[i][j][k];

intgroup int0:
  forall{i,j} where i in (1,n), j in (1,n):
    CB[i] -> CBMAP[i][j];

intgroup int1:
  forall{i,j} where i in (1,n), j in (1,n):
    ORIG[j] and CBMAP[i][j] -> CB[i];

intgroup int1:
  forall{i,j,k,s}
    where i in (1,n), j in (1,n), k in (1,n), s in (1,2):
      CBMAP[i][j] and CLCOMP[j][k][s] -> PROOF[i][k][s];

intgroup int1:
  forall{i,j,k,s}
    where i in (1,n), j in (1,n), k in (1,n), s in (1,2):
      CBMAP[i][j] and not CLCOMP[j][k][s] -> not PROOF[i][k]
        ][s];

intgroup int0:
  forall{i,j,k} where
    i in (1,n), j in (1,n), k in (1,n) and j!=k:
      RES[i] -> PARENT[j][i][1] and PARENT[k][i][2];

intgroup int0:
  forall{i,k} where i in (1,n), k in (1,n):
    RES[i] -> CANCEL[i][k];

```

```

intgroup int1:
  forall{i,j,k,l,m,n} where
    i in (1,n), j in (1,n), k in (1,n), l in (1,n),
    m in (1,2), p in (1,2) and j!=k, m!=p:
    PARENT[j][i][1] and PARENT[k][i][2] and PROOF[j][l][m]
    and PROOF[k][l][p] -> CANCEL[i][l];

intgroup int1:
  forall{i,j,k} where i in (1,n), j in (1,n), k in (1,2):
    not RES[i] -> not PARENT[j][i][k];

intgroup int1:
  forall{i,j,k} where i in (1,n), j in (1,n), k in (1,2):
    PARENT[j][i][k] -> IN[i];

intgroup int1:
  forall{i,j,k} where i in (1,n), j in (1,n), k in (1,2):
    PARENT[j][i][k] -> IN[j];

intgroup int1:
  forall{i,j,k,l,s}
    where i in (1,n), j in (1,n), k in (1,n),
    l in (1,2), s in (1,2) and j!=i:
    PARENT[j][i][l] and PROOF[j][k][s] and not CANCEL[i][k
    ] ->
    PROOF[i][k][s];

intgroup int1:
  forall{i,j,k,l,m,p}
    where i in (1,n), j in (1,n), k in (1,n),
    l in (1,n), p in (1,2) and j!=k, k!=i:
    PARENT[j][i][l] and PARENT[k][i][2] and not PROOF[j][l
    ][p] and
    not PROOF[k][l][p] -> not PROOF[i][l][p];

intgroup int1:
  forall{i,j,s} where i in (1,n), j in (1,n), s in (1,2):
    CANCEL[i][j] -> not PROOF[i][j][s];

```

```

intgroup int1:
  forall{i,j,s} where i in (1,n), j in (1,n), s in (1,2):
    EMPTY[i]  $\rightarrow$  not PROOF[i][j][s];

intgroup int1:
  forall{i} where i in (1,n):
    EMPTY[i]  $\rightarrow$  RES[i];

intgroup wta:
  forall{i,j,s,k}
    where i in (1,n), j in (1,n), s in (1,2), k in (1,2) and
      s!=k:
    PROOF[i][j][s]  $\rightarrow$  not PROOF[i][j][k];

intgroup wta:
  forall{i} where i in (1,n):
    CB[i]  $\rightarrow$  not RES[i];

intgroup wta:
  forall{i,j,k}
    where i in (1,n), j in (1,n), k in (1,n) and j!=k:
    CBMAP[i][j]  $\rightarrow$  not CBMAP[i][k];

intgroup wta:
  forall{i,j,k,l}
    where i in (1,n), j in (1,n), k in (1,n), l in (1,2) and
      k!=j:
    PARENT[j][i][l]  $\rightarrow$  not PARENT[k][i][l];

intgroup wta:
  forall{i,j,n,k,l}
    where i in (1,n), j in (1,n), m in (1,n), k in (1,2),
      l in (1,2) and i!=m:
    PARENT[j][i][k]  $\rightarrow$  not PARENT[j][m][l];

intgroup wta:
  forall{i,j,k,l,m} where

```

```

    i in (1,n), j in (1,n), k in (1,2), l in (1,2), m in
      (1,2) and k!=l:
    PARENT[j][i][l] -> not PARENT[k][i][m];

intgroup wta:
  forall{i,j,k} where i in (1,n), j in (1,n), k in (1,n) and
    j!=k:
    CANCEL[i][j] -> not CANCEL[i][k];

penalties:
  int0 level 0;
  int1 level 1;
  wta level 2;

```

arqprop.sat