XOR VIA SATYRUS

Cássia Francine Novello

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
Setembro de 2012

XOR VIA SATYRUS

Cássia Francine Novello

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.

____
Profa. Priscila Machado Vieira Lima, Ph.D.

____
Prof. Nelson Maculan Filho, D. Habil.

____
Prof. Carlile Campos Lavor, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2012

Novello, Cássia Francine

XOR via SATyrus/Cássia Francine Novello. – Rio de Janeiro: UFRJ/COPPE, 2012.

X, 53 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França

Priscila Machado Vieira Lima

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2012.

Referências Bibliográficas: p. 51 – 53.

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

Agradecimentos

Meus agradecimentos a minha mãe, pai e irmão, por sempre acreditarem em mim. Muito obrigada aos orientadores Felipe França e Priscila Lima, pela orientação e incentivo. E aos amigos, pelo apoio.

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

XOR VIA SATYRUS

Cássia Francine Novello

Setembro/2012

Orientadores: Felipe Maia Galvão França
Priscila Machado Vieira Lima

Programa: Engenharia de Sistemas e Computação

O ou-exclusivo (XOR) é um subproblema muito comum para outros problemas de otimização mais complexos. SATyrus é um ambiente para modelar e resolver problemas de otimização, que utiliza a lógica para definir restrições e a função objetivo, na linguagem chamada SATish. Um mapeamento de fórmulas lógicas em expressões alébricas gera uma função de energia representativa do problema. Entretanto o mapeamento não possui operador para o ou-exclusivo. Este trabalho explora a flexibilidade da linguagem SATish de modo a demonstrar que, usando um modelo base, é possível modelar outros problemas mais complexos. As modelagens na linguagem SATish denominadas CrossWTA, LogWTA, CrossLogWTA, TreeWTA e CrossTreeWTA são estudadas como alternativas de modelagem do XOR, e a modelagem LogWTA é usada como base para um modelo do TSP.

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

XOR VIA SATYRUS

Cássia Francine Novello

September/2012

Advisors: Felipe Maia Galvão França
Priscila Machado Vieira Lima

Department: Systems Engineering and Computer Science

Exclusive-Or (XOR) is a very commom subproblem to more complex optimization problems. SATyrus is an environment to model and solve optimization problems, that uses logic formulae to specify constraints and objective function, in the language called SATish. Logic formulae are mapped into algebraic expressions, generating a energy function that represents the problem. However, this mapping does not include an operator for XOR. This work explores the flexibility of the SATish modeling language, in order to demonstrate that, with the usage of a base model, it is possible to model more complex optimization problems. The models in SATish language called CrossWTA, LogWTA, CrossLogWTA, TreeWTA and CrossTreeWTA are studied as alternatives for modeling XOR, and the LogWTA model is used as base for a TSP model.

Sumário

Lista de Figuras ix

Lista de Tabelas x

1 Introdução 1

1.1 Motivação 2

1.2 Objetivos e Contribuições 3

1.3 Estrutura do documento 3

2 Conceitos básicos e SATyrus 4

2.1 Satisfatibilidade 4

2.2 Fluxo em grafos 6

2.2.1 Modelo linear inteiro para o TSP 7

2.3 SATyrus 8

2.3.1 Modelagem 9

2.3.2 Compilação 11

2.3.3 Resolução 14

2.4 2-XOR como um subproblema 16

3 Estratégias de modelagem 18

3.1 Introdução 18

3.2 CrossWTA 19

3.3 Modelos baseados em fluxo 21

3.3.1 TSP 21

3.3.2 TSP com nós não obrigatórios 24

3.3.3 TSP com nós não obrigatórios e com custo nos nós 27

3.3.4 TSP num grafo incompleto 27

3.4 Modelagem **LogWTA** 28

3.5 Modelagem **CrossLogWTA** 32

3.6 Modelagem **TreeWTA** 36

3.7 Modelagem **CrossTreeWTA** 38

4 Resultados e Discussão **41**
4.1 CrossWTA 42
4.2 TSP baseado em fluxo 43
4.2.1 *Trigger forall* 44
4.2.2 *Trigger exists forall* 45
4.3 Modelagem **LogWTA** 45
4.4 Modelagem **Crossed LogWTA** 45
4.5 Modelagem **TreeWTA** 46
4.6 Modelagem **CrossTreeWTA** 46

5 Conclusões **48**
5.1 Sumário 48
5.2 Trabalhos Futuros 49

Referências Bibliográficas **51**

Lista de Figuras

2.1 Exemplo de caminho num grafo direcionado 6
2.2 Etapas para resolução de um problema no SATyrus 8
2.3 Opções para a etapa da resolução no ambiente. *possibilidades futuras 9

3.1 Representação do modelo TreeWTA para $n = 4$ 37
3.2 Representação do modelo CrossTreeWTA 39

Lista de Tabelas

2.1 Tabela Verdade e quantidade de cláusulas satisfeitas da fórmula 2.1 . 5
2.2 Cada valor $dist[i][j]$ equivale ao elemento D_{ij}, i ∈ (1,num) e j ∈ (1,num) 10
2.3 Exemplo de solução viável para o problema 2-XOR 16

4.1 Exemplo de custos (simétricos) de uma instância do TSP com 6 cidades (nós) 42
4.2 Resultados para o modelo CrossWTA 43
4.3 Exemplo de solução ótima do TSP baseado em fluxo 43
4.4 Exemplos de valores da matriz c numa solução ótima do TSP baseado em fluxo 44
4.5 Arestas penalizadas na solução ótima do TSP baseado em fluxo com *trigger forall* 44
4.6 Resultados para o modelo TSP baseado em fluxo - *trigger forall* . . . 45
4.7 Resultados para o modelo TSP baseado em fluxo - *trigger exists* . . . 45
4.8 Exemplo de resultado para a abordagem *LogWTA* 46
4.9 Tempo em segundos para instâncias da abordagem LogWTA 46
4.10 Exemplo de resultado para a abordagem Cross LogWTA 47
4.11 Tempo em segundos para instâncias da abordagem CrossLogWTA . . 47
4.12 Tempo em segundos para instâncias da abordagem TreeWTA 47

5.1 Comparação das modelagens do problema 2-XOR 48
5.2 Comparação do tempo de resolucao para diferentes modelagens do problema 2-XOR 49
5.3 Comparação das modelagens do problema TSP 49
5.4 Comparação do tempo de resolução para diferentes modelagens do problema TSP 49

Capítulo 1

Introdução

Um ou-exclusivo (XOR), consiste na realização de exatamente uma escolha, dentre n possibilidades. Um ou-exclusivo de duas dimensões, chamado neste trabalho de 2-XOR, consiste na realização de exatamente uma escolha dentre n possibilidades, n vezes, sendo cada escolha diferente das outras. XOR e 2-XOR são subproblemas muito comuns para problemas de otimização mais complexos, como por exemplo o caixeiro-viajante (*Traveling SalesPerson - TSP*), N-rainhas, coloração de grafos, entre outros.

Estes e outros problemas de otimização podem ser modelados como problemas de Satisfatibilidade Máxima (MAX-SAT). Os problemas MAX-SAT são baseados no conceito de Satisfatibilidade Booleana, que consiste em encontrar um conjunto de valores que tornem determinada fórmula lógica verdadeira. Num contexto MAX-SAT, as cláusulas são interpretadas como restrições, e o objetivo é minimizar o número de cláusulas violadas.

SATyrus [1] é um ambiente para modelar e resolver problemas de otimização. Seu diferencial em relação a outros ambientes é sua linguagem de modelagem, SATish, que utiliza fórmulas lógicas para especificar, tanto as restrições, quanto a função objetivo do problema. Além disso, possui operadores de repetição, que permitem a modelagem das restrições lógicas semelhantes de forma concisa.

O ambiente SATyrus é dividido em três etapas: a modelagem, a compilação e a resolução. A primeira versão de SATyrus [2] foi realizada em 2006 e, em 2010, o ambiente foi totalmente reimplementado, dando origem ao SATyrus2 [1]. Uma das motivações para a criação de uma nova versão foi facilitar o uso da linguagem SATish, permitindo a modelagem de restrições com fórmulas bem-formadas na lógica proposicional, diferente da primeira versão, em que a modelagem era restrita a Forma Normal Conjuntiva (FNC).

Outra diferença entre as duas versões está na etapa de resolução. Na primeira versão, a resolução do problema utilizava uma combinação de de Redes de Hopfield [3] com algoritmos de Recozimento Simulado (*Simulated Annealing - SA*)[4] ,

conhecida como Redes de Hopfield Estocásticas [5]. A função objetivo, neste contexto, é chamada de *função de energia*, nomenclatura que permanece sendo usada na segunda versão do ambiente. Entretanto, no SATyrus2, a resolução da função de energia é realizada através da integração do ambiente com softwares externos, ou seja, não é restrita a redes de Hopfield Estocásticas, e torna possível a escolha do resolvedor mais apropriado para o problema.

O mapeamento [6] [7] que transforma o modelo SATish na função de energia representativa do problema ocorre durante a etapa de compilação. O ambiente transforma os operadores lógicos do modelo em expressões algébricas, e cada literal lógico em uma variável matemática, dando origem a função de energia. O problema deve ser modelado de forma que a minimização da função de energia leve a solução ótima.

Qualquer problema que possa ser modelado com restrições é uma possível entrada para o ambiente, por exemplo, problemas de otimização como o TSP e coloração de grafos. ARQ-PROP II, o raciocinador baseado em lógica proposicional, proposto por Lima em [8] [9], também é um possível problema alvo para o ambiente SATyrus. A partir deste ponto, as menções ao ambiente SATyrus referem-se somente a segunda versão do ambiente, SATyrus2.

1.1 Motivação

A especificação para restrições utilizando fórmulas lógicas é flexível, de forma que possibilita a modelagem de problemas mais complexos usando um modelo base, com apenas pequenas adaptações. É possível, por exemplo, usando o modelo do TSP como base, construir um modelo para o problema do caminho mínimo entre dois nós. Outro aspecto interessante da linguagem SATish é a abordagem para problemas que não possuem solução viável. Por exemplo, em uma instância para o problema do TSP que não possui solução viável, o ambiente seria capaz de sugerir a construção de estradas, ou então de indicar o maior caminho que é possível, ainda que não passe em todas as cidades, enquanto outros ambientes apenas responderiam que não existe solução viável.

O mapeamento de Pinkas [6] e Lima [7] não inclui o operador XOR. O raciocinador ARQ-PROP II, assim como o TSP, também possui o 2-XOR como subproblema. Em [1], o TSP é modelado utilizando um modelo do 2-XOR como base. Entretanto, a modelagem para o 2-XOR utilizada como base para o TSP não mostrou-se adequada quando adaptada para ARQ-PROP II. Monteiro [1] conclui isso ao encontrar um mínimo na função de energia que não representa o mínimo esperado para o problema. A criação de alternativas para a modelagem do XOR e 2-XOR tornou-se importante para a continuação dos estudos utilizando o ambiente SATyrus.

1.2 Objetivos e Contribuições

Os objetivos deste trabalho são (i) a exploração da flexibilidade da linguagem SATish, permitindo a modelagem de variações de um problema realizando pequenas alterações em um modelo base; (ii) a construção de modelagens alternativas para o XOR e 2-XOR na linguagem SATish, uma vez que o mapeamento de Pinkas e Lima não inclui o operador XOR; (iii) a utilização dos modelos alternativos para o XOR como base para a modelagem de um problema mais complexo: o TSP.

Para a exploração da flexibilidade da linguagem, foi utilizado um modelo para o TSP que utiliza fluxos em grafos, baseado no modelo matemático de Lisser, Plateu e Maculan [10]. As variações exploradas incluem TSP com nós não obrigatórios, TSP com nós não obrigatórios e custos nos nós, e TSP num grafo incompleto, possivelmente sem solução viável.

Para a modelagem do XOR e 2-XOR foram investigadas cinco alternativas: CrossWTA [11] [6], LogWTA [12], TreeWTA [11], CrossLogWTA [13] e CrossTreeWTA, sendo esta última, introduzida neste trabalho na seção 3.7. Todas foram modeladas na linguagem SATish e resolvidas através da integração do SATyrus com o resolvedor Bonmin [14], por intermédio da linguagem AMPL [10]. Os modelos são comparados quanto ao número de variáveis, número de restrições, e tempo médio para encontrar a solução ótima, dados os parâmetros adequados.

A modelagem LogWTA foi utilizada como base para o TSP. A modelagem do TSP baseada em CrossWTA, anterior a este trabalho, é comparada com a nova modelagem, com base em LogWTA, assim como as modelagens para o TSP baseadas em fluxo, em relação ao tempo médio de resolução.

1.3 Estrutura do documento

Este documento divide-se em cinco capítulos. O Capítulo 2 apresenta os fundamentos teóricos para este trabalho. Ele inclui os conceitos de satisfatibilidade e satisfatibilidade máxima, fluxo em grafos voltado para a modelagem do TSP, o ambiente SATyrus, sua linguagem e funcionamento, e a definição dos problemas XOR e 2-XOR no contexto da otimização via SATyrus.

O Capítulo 3 apresenta os modelos referentes ao TSP baseado em fluxo e suas variações, os cinco modelos alternativos para o 2-XOR, bem como a modelagem do TSP baseado na alternativa LogWTA. O Capítulo 4 mostra a média do tempo de resolução para os modelos alternativos do XOR, e para os modelos do TSP, e exemplos de solução ótima em cada modelo.

Finalmente, o Capítulo 5 apresenta comparações do número de variáveis, restrições e tempo médio para resolução, conclusões e sugestões para trabalhos futuros.

Capítulo 2

Conceitos básicos e SATyrus

2.1 Satisfatibilidade

Satisfatibilidade Booleana (conhecido como SAT) é um problema de decisão que consiste em verificar se um conjunto de valores tornam uma determinada fórmula lógica verdadeira. Se não existir tal conjunto de valores, a fórmula é dita instatisfatível. Vários tipos de problemas de decisão e otimização podem ser modelados como problemas de satisfatibilidade booleana, como coloração de grafos, caixeiro viajante (*Traveling Salesman Problem* - TSP) [1], predição das conformações moleculares estáveis (*Stable Molecular Conformations Prediction Problem* - SMCPP) [15] e planejamento de expansão de geração de energia (*Generation Expansion Planning Problem* - GEP) [16].

O problema de satisfatibilidade booleana pertence à classe NP-completo [17], isto é, dada uma possível solução, ela pode ser verificada com complexidade de tempo polinomial. Entretanto, o tempo para verificar uma solução cresce conforme o tamanho do problema. Além disso, não existem algoritmos rápidos conhecidos que localizem, dentre as possíveis soluções, aquela(s) que torna(m) determinada fórmula verdadeira.

Neste trabalho, são denominadas fórmulas, as fórmulas bem-formadas na lógica proposicional. Elas devem seguir as seguintes leis de formação:

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

Um caso particular das fórmulas bem-formadas é a Forma Normal Conjuntiva (FNC) [18]. Neste formato, cada fórmula é uma conjunção de cláusulas e cada

cláusula é uma disjunção de literais. Assim, apenas os operadores lógicos $\wedge$, $\vee$ e $\neg$ podem ser utilizados e os outros devem ser convertidos para estes.

Por exemplo, considere a fórmula lógica bem-formada$(p \rightarrow q) \wedge r$. Convertendo para o formato FNC, obtemos a fórmula 2.1, com três literais (p, q e r) e duas cláusulas ($\neg p \vee q$ e r).

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

A *tabela verdade* de uma fórmula lógica enumera cada possível valoração para cada um dos literais. Podemos ver na Tabela 2.1 que três valorações para p, q e r satisfazem a fórmula. Entretanto, avaliar cada uma das possíveis valorações torna-se mais difícil conforme a quantidade de literais e cláusulas aumenta.

Tabela 2.1: Tabela Verdade e quantidade de cláusulas satisfeitas da fórmula 2.1

p	q	r	$(\neg p \vee q)$	$(\neg p \vee q) \wedge r$	Cláusulas satisfeitas
V	V	V	V	**V**	2
V	V	F	V	F	1
V	F	V	F	F	1
V	F	F	F	F	0
F	V	V	V	**V**	2
F	V	F	V	F	1
F	F	V	V	**V**	2
F	F	F	V	F	1

O problema da Satisfatibilidade Máxima (MAX-SAT) é um problema de otimização baseado no conceito de satisfatibilidade. Consiste em, dada uma fórmula na Forma Normal Conjuntiva, buscar uma valoração que maximize o número de cláusulas satisfeitas.

Pode-se associar o problema MAX-SAT a um problema de satisfação de restrições, considerando as cláusulas como restrições. Neste caso, o processo de resolução busca diminuir o número de restrições violadas. Frequentemente, são usados algoritmos aproximativos ou heurísticos para explorar possíveis soluções e buscar a solução ótima.

Pode-se associar também, problemas MAX-SAT a problemas de programação inteira 0-1 [6] [7], devido a semelhança de sua natureza binária. 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. Um mapeamento para esta associação será descrito com detalhes em 2.3.2.

2.2 Fluxo em grafos

Em um grafo $G = (V, E)$, V é conjunto finito não vazio, e E é um conjunto de pares não-ordenados de elementos distinos de V [19]. Os elementos de V são denominados nós ou vértices e os elementos de E são denominados arestas.

Seja e=$(v,w) \in E$,os nós v e w são ditos adjacentes ou vizinhos, e extremidades da aresta e. A quantidade de nós adjacentes a v é chamada de grau do nó v. No caso em que $v = w$, a aresta $e = (v, v)$ é denominada laço.

Uma sequência v_1, ... , v_k é chamada de caminho de v_1 a v_k, onde (v_j, v_{j+1}) são adjacentes para cada j tal que $1 \leq j \leq k-1$. Um caminho de k nós é formado por $k-1$ arestas $(v_1, v_2), (v_2, v_3), ...(v_{k-1}, v_k)$. É chamada de distância entre dois nós d(v,w), o comprimento do menor caminho entre v e w.

Um grafo $G = (V, E)$ é dito conexo quando existe um caminho entre cada par de nós, e desconexo caso contrário. Um caminho v_1, ... v_k onde $v_1 = v_k$ é denominado ciclo. Um caminho que contenha cada nó do grafo exatamente uma vez é denominado *Hamiltoniano*.

Um grafo conexo e acíclico (que não contém ciclos), é denominado árvore. Em uma árvore, nós de grau 1 são chamados de folhas. Os nós de grau > 1 são chamados nós internos. Um determinado nó pode ser escolhido para raíz da árvore. A distância entre um nó qualquer e a raíz é chamada nível do nó.

Um grafo é dito completo quando existe uma aresta entre cada par de seus nós. Um subgrafo de $G = (V, E)$ é $G_1 = (V_1, E_1)$ quando $G_1 \supseteq G$ e $E_1 \supseteq E$.

Um grafo direcionado é um conjunto finito não vazio G e um conjunto de pares **ordenados** E, de elementos distinos de V. Ou seja, em um grafo direcionado, cada aresta (v, w) possui uma única direção de v para w, de forma que $(v, w) \neq (w, v)$. Todas as definições anteriores também são válidas para o caso de grafos direcionados.

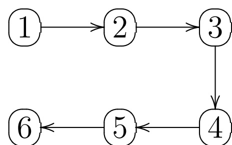

Figura 2.1: Exemplo de caminho num grafo direcionado

Grau de entrada é a quantidade de arestas que chegam a v, e grau de saída é a quantidade de arestas que partem de v. Nós que possuem grau de entrada = 0 são chamados de origem ou fonte, e nós com graus de saída = 0 são chamados de sumidouro.

O fluxo em um grafo é a modelagem da transferência de algum recurso quantificável [20]. Isto é, quando o recurso é transferido de um nó para outro, deixa de existir no primeiro e passa a existir no segundo. É possível uma analogia com o tráfego urbano, onde as ruas são as arestas, os nós são os cruzamentos, e os carros

são os recursos.

Um grafo com fluxo é representado por $G = (V, E, f)$, onde os elementos do vetor f indicam o valor do fluxo em uma aresta e. Pela lei de conservação do fluxo, a quantidade de fluxo que entra em um nó, v deve ser igual a quantidade de fluxo que sai do nó v. Pode-se definir também a capacidade de fluxo constante c para cada aresta j, tal que $f_j < c$ ou ainda um vetor c_j tal que $f_j < c_j$.

2.2.1 Modelo linear inteiro para o TSP

Um modelo linear inteiro para o TSP [10] é baseado problema de fluxo não simultâneo. Seja $G = (V, E)$ um grafo conexo, onde $V = \{1, 2, 3, ..., n\}$ é um conjunto de nós e E um conjunto de arestas. Seja Gd um grafo direcionado derivado de G onde $A = \{(i, j), (j, i) | \{i, j\} \in E\}$, isto é, cada aresta u = {i,j} ∈ E está associada com dois arcos (i,j) e (j,i) ∈ A. Seja $y = (y_u)_{u \in E} \in \{0, 1\}^{|E|}$ um vetor 0-1 e $z_{ij}^k \geq 0 (i, j) \in A$, $k \in V'$, onde V' é um subconjunto de V, e z_{ij}^k é um fluxo real no arco (i,j) ∈ A, tendo qualquer nó escolhido como a origem, e k como o sumidouro. Considera-se E(i) um conjunto de arestas u ∈ E que terminam em i, $\Gamma^+(i) = \{j | (i, j) \in A\}$ e $\Gamma^-(i) = \{j | (j, i) \in A\}$. Números reais associados a w_u representam os custos, distâncias, pesos, etc. Cada vetor y que satisfaz as restrições (2.2-2.9) é associado com um ciclo Hamiltoniano mínimo de G.

$$\sum_{j \in \Gamma^+(1)} z_{1j}^k - \sum_{j \in \Gamma^-(1)} z_{j1}^k = 1, k \in V - \{1\} \tag{2.2}$$

$$\sum_{j \in \Gamma^+(i)} z_{ij}^k - \sum_{j \in \Gamma^-(i)} z_{ji}^k = 0, i \in V - \{1, k\}, k \in V - \{1\} \tag{2.3}$$

$$\sum_{j \in \Gamma^+(k)} z_{kj}^k - \sum_{j \in \Gamma^-(k)} z_{jk}^k = -1, k \in V - \{1\} \tag{2.4}$$

$$z_{ij}^k \leq y_{ij} and z_{ji}^k \leq y_{ij}, \{i, j\} \in E, k \in V - \{1\} \tag{2.5}$$

$$\sum_{u \in E(i)} y_u = 2, i \in V \tag{2.6}$$

$$z_{ij}^k \geq 0, (i, j) \in A, k \in V - \{1\} \tag{2.7}$$

$$y_{ij} \in \{0, 1\}, (i, j) \in E \tag{2.8}$$

$$minimize \sum_{u \in E} w_u y_u \tag{2.9}$$

2.3 SATyrus

SATyrus [21] é um ambiente para modelagem e resolução de problemas de otimização, vide Figura 2.2. Sua linguagem de modelagem, SATish, utiliza a lógica para expressar as restrições e função objetivo do problema alvo. Em seguida, ocorre a compilação, que substitui operadores lógicos por subexpressões algébricas, compondo uma função de energia representativa do problema [22]. A função de energia é passada a um resolvedor externo, que irá buscar a solução ótima.

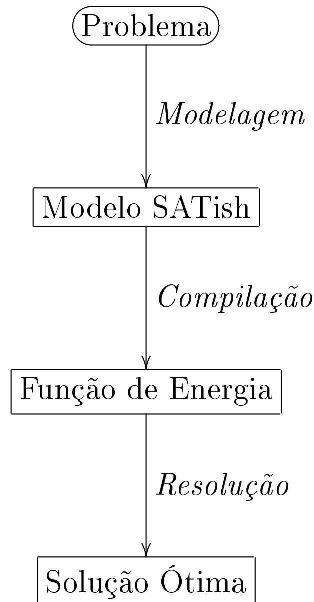

Figura 2.2: Etapas para resolução de um problema no SATyrus

A primeira versão do SATyrus [2] foi realizada em 2006, implementada em C++. Nesta versão, a definição de restrições utiliza fórmulas na FNC e a etapa da resolução utiliza uma combinação de Redes de Hopfield [3] com algoritmos de Recozimento Simulado (*Simulated Annealing* – SA) [23] [4], conhecida como Redes de Hopfield Estocásticas [5].

Em 2010, o ambiente foi totalmente reimplementado na linguagem Python, dando origem ao SATyrus2 [1]. Uma das motivações para a criação da nova versão do ambiente SATyrus foi consolidar a linguagem de especificação de problemas, SATish. A nova versão veio aprimorar a linguagem original possibilitando a escrita de modelos mais concisos e facilitando a modelagem de restrições, utilizando fórmulas bem-formadas em vez do padrão FNC.

Outra motivação para a construção da nova versão do ambiente foi modularizar o código, de forma a possibilitar o uso de alternativas para a etapa da resolução. Na primeira versão do ambiente a etapa da resolução dava-se apenas através da criação de redes de Hopfield Estocásticas. Já no SATyrus2, um submódulo da interface expressa a função de energia gerada na linguagem de modelagem do software externo

escolhido pelo usuário, e cada um deles possui diversos resolvedores a sua disposição. Os softwares externos já integrados com o SATyrus2 são AMPL [24] e Xpress [25], que usam as linguagens AMPL e Mosel, respectivamente. Devido a arquitetura modular, é possível a criação de interfaces com mais plataformas de otimização no futuro.

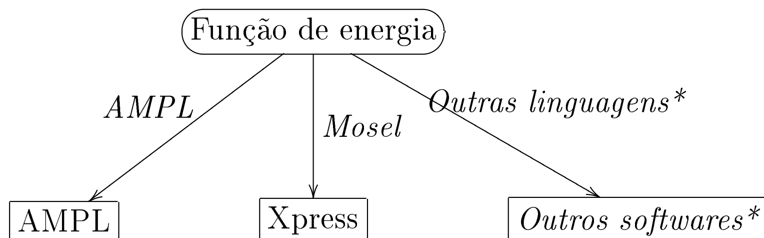

Figura 2.3: Opções para a etapa da resolução no ambiente. *possibilidades futuras

2.3.1 Modelagem

A modelagem na linguagem SATish possui três etapas de definição: constantes e estruturas, restrições e penalidades. Constantes e estruturas são identificadores. Restrições definem o problema em si, utilizando a lógica proposicional. Penalidades atribuem níveis de importância às restrições.

As constantes são identificadores usados para representar um número inteiro, facilitando a legibilidade e adaptações no modelo. Em SATish, as constantes podem ser utilizadas em operaçoẽs aritméticas, definição de estruturas e intervalos. Entretanto, não podem ser usadas como índice na definição de uma restrição.

As estruturas são identificadores indexados, usados para representar matrizes, ou seja, um conjunto de valores correlacionados. A definição do identificador de uma estrutura deve ser seguida de suas dimensões. Os índices de uma estrutura representam a posição de um elemento em uma matriz multidimensional. Índices são uma sequencia iniciada em 1 e limitada pela dimensão da estrutura.

Os valores da estrutura pode ser considerados constante numéricas ou variáveis proposicionais. Constantes numéricas são pré-definidas no modelo, e podem assumir qualquer valor inteiro. Já variáveis proposicionais podem ou não ser pré-definidas e, naturalmente, assumem apenas os valores 0 ou 1.

Na linha 1 da Listagem 2.1, definimos a constante *num*, que irá representar o número 2. Nas linhas 2 e 3, definimos as estruturas *pos* e *dist*, respectivamente, utilizando a constante *num*. Os valores da estrutura *dist* são constantes numéricas, pré-definidos nas linhas 4-7. Os valores da estrutura *pos* são variáveis proposicionais do problema alvo. Na Tabela 2.2, a estrutura *dist* pode ser visualizada como uma matriz D, onde o valor dist[1][1] equivale ao elemento D_{11}, $dist[1][2]$ ao elemento D_{12}, e assim por diante.

```
num=2;
pos(num,num);
dist(num,num);
dist = [
1,1:5; 1,2:4;
2,1:3; 2,2:7
];
```

Listagem 2.1: Exemplo da definição de estruturas na linguagem SATish

Tabela 2.2: Cada valor $dist[i][j]$ equivale ao elemento D_{ij}, i ∈ (1,num) e j ∈ (1,num)

D	*1*	*2*
1	5	4
2	3	7

As restrições representam as propriedades do problema alvo. São representadas por qualquer fórmula bem-formada da lógica proposicional. Podem ser de dois tipos: viabilidade (ou integridade, representada pela palavra reservada *intgroup*) e otimalidade (representadas pela palavra reservada *optgroup*). Restrições de viabilidade definem o espaço de busca de soluções viáveis, de forma que o valor da função de energia diminua a medida que mais restrições deste tipo são satisfeitas. Já as restrições de otimalidade atribuem custos para as soluções viáveis, de forma que o valor da função de energia acompanhe os custos das soluções.

É possível modelar restrições semelhantes utilizando os repetidores *forall* ou *exists*, para deixar o modelo mais conciso. Neste caso, é necessário definir os índices e intervalos para os quais a restrição deve ser válida. *forall* representa a validade da restrição para cada um os índices definidos no intervalo, enquanto *exists* representa a validade da restrição para pelo menos um dentre os índices definidos no intervalo.

Cada restrição deve estar associada a um nível de penalidade, sendo que várias podem ser associadas ao mesmo nível. A penalidade pode ser considerada o nível de importância da restrição, ou seja, quanto maior o nível de penalidade de uma restrição, maiores as chances de ela ser satisfeita no processo de resolução.

Temos um exemplo de restrição na Listagem 2.2. Na linha 1, vemos que trata-se de uma restrição de viabilidade associada ao nível de penalidade wta. Na linha 2, temos o repetidor $forall$, que replica a restrição para todos os índices e intervalos definidos, e a condição $j \neq l$, que define exceções ao repetidor. Na linha 3, finalmente, temos uma fórmula bem-formada que combina valores da estrutura *pos*, modelando as propriedades do problema.

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

Listagem 2.2: Exemplo da definição de restrição na linguagem SATish

Considerando $num = 2$ já é possível observar a vantagem de utilizar o repetidor $forall$. A Listagem 2.2 mostra uma forma compacta para a modelar que todas as seguintes restrições devem ser satisfeitas:

$$pos_{11} \rightarrow \neg pos_{12}$$

$$pos_{12} \rightarrow \neg pos_{11}$$

$$pos_{21} \rightarrow \neg pos_{22}$$

$$pos_{22} \rightarrow \neg pos_{21}$$

Cada nível de penalidade está associado a um número inteiro positivo. As restrições de otimalidade devem ser associadas a níveis menores do que as de viabilidade. A definição dos níveis de penalidade conclui a modelagem do problema.

```
penalties:
  wta level 2;
  int1 level 1;
  costo level 0;
```

Listagem 2.3: Exemplo da definição de penalidades na linguagem SATish

A linguagem permite comentários entre /* e */ ou após //, para ajudar na compreensão do modelo.

2.3.2 Compilação

SATyrus2 atua também como compilador da linguagem SATish. O compilador foi construído utilizando o PLY (Python Lex-Yacc), uma implementação grátis, totalmente em Python das ferramentas já amplamente conhecidas para construção de compiladores: Lex e Yacc. SATyrus2 recebe como entrada o modelo do problema alvo na linguagem SATish, e produz como saída a função de energia, na linguagem escolhida pelo usuário do ambiente.

A primeira tarefa do compilador é identificar erros na modelagem. Alguns erros possíveis são: “*index out of bounds*”, pelo menos um índice se encontra fora dos

limites dimensionais definidos, e "*wrong number of dimensions for struct X*", quando a estrutura é referênciada com uma quantidade de índices diferente da sua definição.

Após a eliminação dos erros de modelagem, SATyrus2 converte automaticamente as fórmulas bem-formadas do modelo SATish para a Forma Normal Conjuntiva, de forma a permitir o mapeamento que associa um problema de satisfatibilidade booleana a um problema de programação linear inteira 0-1. Este mapeamento, criado por Pinkas [6] e aperfeiçoado por Lima et al. [7], constrói uma função objetivo E, neste contexto denominada função de energia, que representa o espaço de busca do problema alvo.

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. Durante o mapeamento, cada literal p_{ij} dá origem a uma única variável matemática, onde o valor booleano *verdadeiro* é associado com o valor numérico 1, e o valor *falso* é associado a 0. A fórmula φ é mapeada em expressões algébricas, de acordo com as regras a seguir:

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

A função de energia E, resultado do mapeamento, pode assumir o valor 0 ou 1, de acordo com a valoração das variáveis matemáticas. Contudo, é possível adaptar a função H, de modo a transformar a expressão obtida pelo mapeamento em uma função para um problema de otimização. Caso deseje-se minimizar a função, como é o caso no SATyrus2, deve-se aplicar a negação a fórmula φ antes do mapeamento, obtendo $\neg\varphi = \bigvee \neg\varphi_i$.

Por exemplo, deseja-se obter um problema de minimização associado a fórmula 2.10, de apenas uma cláusula:

$$\phi = p \vee \neg q \tag{2.10}$$

Interpretando através da satisfatibilidade, deseja-se encontrar uma valoração que minimize o número de cláusulas não satisfeitas. Interpretando através da programação linear inteira 0 -1, deseja-se obter uma valoração que minimize a função objetivo 2.11. Caso uma solução com custo igual a 0 seja encontrada, a fórmula ϕ é

satisfeita. A seguir, a aplicação das regras de mapeamento na fórmula ϕ:

$$\begin{aligned} E &= H(\neg\phi) \\ &= H(\neg(p \vee \neg q)) \\ &= H(\neg p \wedge q) \\ &= H(\neg p) \times H(q) \\ &= \underbrace{(1 - H(p)) \times H(q)}_{\text{função de energia}} \end{aligned} \tag{2.11}$$

Uma adaptação para o problema MAX-SAT é possível, utilizando uma função somatório $H^{\star}$, capaz de contar as cláusulas não satisfeitas, dada uma valoração qualquer. Substituímos a equação $E = H(\neg\varphi)$ por $E = H^{\star}(\neg\varphi) = \sum_i H(\neg\varphi_i)$. Portanto, $E = \sum_i H(\bigwedge_j \neg p_{ij}) = \sum_i \prod_j H(\neg p_{ij})$. É interessante notar que uma função de energia obtida desta forma possui a característica de ser formada apenas por variáveis matemáticas de primeira ordem.

Os níveis de penalidade de cada restrição, expressados na definição do problema na linguagem SATish, são transformados em pesos para as parcelas da função de energia. Os pesos representam um acréscimo na função de energia, caso a restrição associada a ele seja violada. As penalidades são calculadas de modo a tornar mais vantajosa a satisfação de cláusulas que pertençam a níveis de penalidade mais elevados. Seja n_i a quantidade total de cláusulas dentre todas as restrições de nível i, e seja ϵ um valor pequeno. As penalidades (v_i) associadas aos diferentes níveis são calculadas da seguinte forma:

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

A penalização por violar todas as restrições de um nível i é menor do que a penalização por violar apenas uma restrição do nível $i{+}1$. Mais especificamente, para um nível i qualquer, seja v_i o valor numérico da penalidade associada a i, o cálculo do valor da penalidade associada o nível $i + 1$ possui a parcela $((n_i + 1) \times v_i) + \epsilon$. 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 $((n_i + 1) \times v_i) + \epsilon$.

2.3.3 Resolução

SATyrus2 pode expressar a função de energia nas linguagens AMPL, ou Mosel, dos softwares AMPL [24] e Xpress [25], respectivamente. O usuário é responsável por apresentar os arquivos gerados para o resolvedor escolhido. SATyrus2 possui uma arquitetura modular que facilita a futura integração do ambiente com outros resolvedores.

Neste trabalho, foi explorada a integração com o software AMPL e o resolvedor BONMIN (Basic Open-source Nonlinear Mixed INteger programming) [14] . Dentre os seis possíveis algoritmos do BONMIN, o algoritmo padrão B-BB (branch-and-bound) foi escolhido, por ser o mais indicado para tratar MINLPs (*Mixed Integer NonLinear Programming*) não convexos. Para cada modelo, SATyrus2 gera dois arquivos em AMPL: o arquivo com a extensão *.in*, que contém os parâmetros para o resolvedor escolhido, vide Listagem 2.4, e outro com a extensão *.mod*, que contém a função de energia em si, vide Listagem 2.5.

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

Listagem 2.4: Arquivo exemplo.in na linguagem AMPL

```
# Variables:
var y_2_2 binary;
var y_2_3 binary;
var y_2_1 binary;
var y_1_1 binary;
var y_1_3 binary;
var y_1_2 binary;

# Energy Function:
minimize f: 1.000000 * ((
(y_1_2 * y_1_1) + (y_1_3 * y_1_1) + (y_1_1 * y_1_2) +
(y_1_3 * y_1_2) + (y_1_1 * y_1_3) + (y_1_2 * y_1_3) +
(y_2_2 * y_2_1) + (y_2_3 * y_2_1) + (y_2_1 * y_2_2) +
(y_2_3 * y_2_2) + (y_2_1 * y_2_3) + (y_2_2 * y_2_3)
));
```

Listagem 2.5: Arquivo exemplo.mod na linguagem AMPL

O algoritmo BB divide o espaço de busca entre subespaços que contém soluções em potencial e aqueles que não contém. Sendo assim, os subespaços descartados são

explorados apenas implicitamente, enquanto os potenciais seguem sendo explorados, até encontrar a solução ótima. Os subespaços são representados por limites superior e inferior, sendo o limite inferior obtido através da resolução de subproblemas, e o limite superior obtido através de soluções viáveis. Os limites são ajustados durante a execução do algoritmo. Cada subespaço é um nó da árvore de busca.

O descarte dos nós se dá pela combinação dos limites representativos dos subespaços com valor da solução ótima conhecida até o momento. Inicialmente, a solução ótima conhecida é ∞ e o único subespaço é o espaço de busca completo, que é representado como a raíz da árvore de busca. Um nó pode ser descartado *(fathomed)* porque:

- o subproblema é inviável;
- seu limite inferior é maior do que o limite superior do n que contém a solução ótima atual;
- a resolução do subproblema encontra uma solução viável inteira;

Por outro lado, se for encontrada uma solução viável fracionária, o nó é explorado. O algoritmo segue até que todos os nós tenham sido descartados ou explorados, e a solução ótima encontrada é considerada a solução ótima do problema como um todo. A exploração de um nó possui três etapas:

1. Seleção do nó a ser explorado;
2. Cálculo dos limites do subespaço (*Bound*);
3. Criação dos subespaços filhos (*Branch*).

Cada uma destas etapas pode ser implementada de diversas formas. A implementação padrão do BB no resolvedor BONMIN realiza a seleção do próximo nó a ser explorado utilizando a estratégia *best-bound*, ou seja, aquele que possui menor limite inferior. A criação dos subespaços filhos utiliza pseudo-custos, inicializados com a média de pseudo-custos conhecidos, e *strong branching*. Estes e outros detalhes sobre a implementação do BONMIN encontram-se em [26]. Mais detalhes sobre os parâmetros padrão e outras possibilidades para o BONMIN encontram-se em [14].

SATyrus2 invoca o algoritmo BB padrão, através da criação do arquivo de extensão *.mod*. É possível adicionar os parâmetros do resolvedor através da alteração dos arquivos gerados pelo ambiente, antes de apresentá-los ao resolvedor. Os parâmetros *resolve_at_rootk* e *resolve_at_nodek*, por exemplo, determinam k pontos iniciais aleatórios para explorar a raíz, ou todos os nós da árvore de busca, respectivamente, e guardam a melhor solução encontrada. Mais detalhes sobre o a utilização destes parâmetros encontram-se no Capítulo 4.

2.4 2-XOR como um subproblema

Um ou-exclusivo, XOR, consiste na realização de uma escolha, dentre n possibilidades. Um ou-exclusivo de duas dimensões, 2-XOR, consiste na realização de uma escolha dentre n possibilidades, n vezes, sendo que cada escolha deve ser diferente das outras. O 2-XOR também pode ser definido como o posicionamento de n peças em uma estrutura nxn, sendo que cada peça não pode compartilhar linhas nem colunas com nenhuma das outras peças. Existem $n!$ soluções viáveis distintas para este problema, uma vez que são n possibilidades para a primeira escolha, $n-1$ para a segunda, $n-2$ para a terceira, e assim por diante, dentre as 2^{nxn} possibilidades irrestritas.

XOR e 2-XOR são subproblemas muito comuns para outros problemas maiores. O problema do posicionamento das N-Rainhas, por exemplo, pode ser modelado utilizando o 2-XOR como base, e acrescentando restrições para as peças não compartilharem diagonais. O TSP também pode usar o 2-XOR como base para a sua modelagem, acrescentando as restrições de custos do caminho entre cidades vizinhas. Neste caso, podemos interpretar uma dimensão como cidades, e a outra dimensão como posições de cada cidade no caminho. Uma solução viável é definida pela associação de uma posição distinta para cada cidade no caminho.

Tabela 2.3: Exemplo de solução viável para o problema 2-XOR

cidades/posições	1	2	3	4	5
Natal	1				
Rio de Janeiro					1
São Paulo				1	
Belo Horizonte			1		
Salvador		1			

A modelagem lógica do XOR, a princípio, necessita de operadores existenciais, ou seja, dada uma disjunção de cláusulas, deve existir apenas uma verdadeira. Como o mapeamento criado por Pinkas não possui operador para o XOR, é utilizada em [27] e [1], uma combinação de restrições para eliminar os existenciais e assegurar as disjunções exclusivas. Este modelo, agora denominado CrossWTA, será reapresentado no capítulo 3.

O 2-XOR também pode ser utilizado como subproblema da modelagem de ARQ-PROP II [9], uma máquina de raciocínio lógico. ARQ-PROP II é capaz de realizar inferências utilizando uma base de conhecimento ou incompleto, ou seja, também permite hipotetizar cláusulas para esta base, utilizando o Princípio da Resolução aplicado a lógica proposicional. Em [1], de Monteiro, encontra-se a modelagem SATish para ARQ-PROP II utilizando o 2-XOR CrossWTA como base, .

Ao resolver ARQ-PROP II utilizando o ambiente SATyrus2, Monteiro encontrou

um mínimo global que aponta uma falha na modelagem, uma vez que a representação da solução ótima esperada está associada a uma energia maior do que o mínimo encontrado pelo resolvedor. Mostrou-se então que a modelagem 2-XOR CrossWTA garante de fato a validade do ou-exclusivo, mas quando utilizada com subproblema para problemas maiores, pode gerar dificuldades. No próximo capítulo serão discutidas as causas dessas dificuldades, e apresentadas outras alternativas para a modelagem do 2-XOR na linguagem SATish.

Capítulo 3

Estratégias de modelagem

3.1 Introdução

Utilizando um modelo base é possível obter modelos para variações do problema, realizando pequenas alterações, como por exemplo, a inclusão ou a exclusão de um grupo de restrições. Os modelos baseados em fluxo podem demonstrar a flexibilidade da linguagem SATish. O problema TSP vem sendo utilizado para testes no SATyrus desde a criação do ambiente, em 2006. Os modelos baseados em fluxo utilizam o TSP como base e algumas das variações são descritas a seguir.

- TSP - o problema clássico, em que deve-se encontrar o caminho de custo mínimo para visitar n cidades, retornando a cidade de origem. Todas as cidades possuem estradas entre si, ou seja, o grafo é comlpeto. Os custos são simétricos, ou seja, o custo da viagem da cidade X para a cidade Y é o mesmo de Y para X. Também conhecido como Ciclo Hamiltoniano Mínimo.

- TSP com nós não obrigatórios - Adiciona-se uma estrutura capaz de determinar quais nós devem fazer parte da solução e quais podem ou não fazer parte. O caso mais simples desta variação é o problema do caminho mínimo entre dois nós, em que apenas dois nós são obrigatórios na solução.

- TSP com nós não obrigatórios e com custo nos nós - Adiciona-se uma restrição que contabiliza os pesos dos nós, assim como os pesos das arestas.

- TSP com arestas inexistentes - caso o grafo de origem não seja completo, há três possibilidades:

  - as arestas inexistentes podem sem criadas para permitir a formação do ciclo hamiltoniano mínimo passando por todos os nós.
  - as arestas inexistentes não podem ser criadas e deve-se achar o maior ciclo hamiltoniano possível.

– uma combinação das duas acima.

O 2-XOR pode ser considerado um subproblema do TSP, em que deve-se encontrar um caminho, onde cada cidade ocupa apenas uma posição deste caminho (primeira dimensão), e não há repetição de cidades (segunda dimensão). Os modelos **CrossWTA**, **LogWTA** [12], **CrossLogWTA**, **TreeWTA** e **CrossTreeWTA** são alternativas para a modelagem do 2-XOR e também podem ser utilizados para a posterior modelagem do TSP.

Alguns tipos de restrições são comuns a vários modelos e por isso recebem denominações específicas. As restrições *trigger*, ou acionadoras, são aquelas que penalizam a solução nula. Serão discutidos dois tipos de triggers:

- Trigger utilizando o repetidor *forall*, representa que a restrição será válida para todos os índices definidos no intervalo;
- Trigger utilizando o repetidor *exists*, representa que a restrição será válida para pelo menos um dos índices definidos no intervalo.

As restrições *Winner Takes All - WTA* são aquelas no formato $F_i \rightarrow \neg F_j$ para $i \neq j$. F pode representar qualquer fórmula bem formada. Elas são utilizadas para assegurar a exclusividade de cada escolha no modelo XOR.

3.2 CrossWTA

O modelo apresentado na Listagem 3.1 para o TSP utiliza a abordagem CrossWTA. Esta denominação deve-se a utlização de restrições WTA para duas dimensões do problema.

As restrições são:

- *Trigger forall*;
- Cada cidade pode ocupar apenas uma posição do caminho (WTA);
- Cada posição do caminho pode conter apenas uma cidade (WTA);
- O custo do caminho deve ser minimizado.

Possui duas estruturas:

pos_{ij} =1, representa que a cidade i ocupa a posição j no caminho;

w_{ij} =k, representa o custo k da viagem entre a cidade i na posição l e j na posição l+1 no caminho. No problema clássico, os custo são simétricos.

```
num=5;
pos(num,num);
dist(num,num);

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 int1: pos[1][1];
intgroup int1: pos[5][5];

// Trigger
intgroup int1:
  forall{i,j} where i in (1,num), j in (1,num):
    pos[i][j];

// Duas cidades não podem ocupar a mesma posição no caminho
intgroup wta:
  forall{i,j,k} where i in (1,num), j in (1,num), k in (1,
     num) and i != k:
    not pos[i][j] or not pos[k][j];

// Uma cidade não pode ocupar mais de uma posição no caminho
intgroup wta:
  forall{i,j,l} where i in (1,num), j in (1,num), l in (1,
     num) and j != l:
    not pos[i][j] or not pos[i][l];

// A soma das distâncias entre as cidades do caminho deve ser minimizada
optgroup costo:
  forall{i,j,k} where i in (1,num), j in (2,num), k in (1,
     num) and i != k:
    dist[i][k] (pos[i][j] and pos[k][j-1]);
optgroup costo:
```

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

penalties:
  wta level 2;
  int1 level 1;
  costo level 0;
```

Listagem 3.1: Modelo SATish para o TSP - abordagem **CrossWTA**

Este modelo é eficiente para resolver o problema do TSP. Entretanto, não mostra-se flexível para permitir variações. No problema do caminho mínimo, por exemplo, em que deseja-se encontrar o menor caminho entre dois nós, o número de cidades que farão parte da solução não é conhecido previamente. Outras modelagens foram testadas com o objetivo de criar um modelo mais flexível.

3.3 Modelos baseados em fluxo

O novo modelo TSP é inspirado na modelagem matemática baseada em fluxo unário, não simultâneo [10], mostrado em 2.2.1. Este modelo será utilizado como base para a modelagem de variações do problema do TSP.

3.3.1 TSP

O modelo TSP baseado em fluxo utiliza três estruturas:

y_{ij} =1, representa que há fluxo no arco ij;

c_{ij} =1, representa que o nó i precede o nó j no caminho. Naturalmente, $c_{ji} = 0$;

w_{ij} =k, representa o custo k da viagem entre ij no caminho.

As restriçõs envolvendo a estrutura y representam o fluxo unário não simultâneo em cada nó. Ou seja, há apenas uma aresta ij e apenas uma aresta jk, para cada nó j, exceto a origem e sumidouro. No modelo, a origem e sumidouro são representados respectivamente pelo primeiro e o último nó.

Para i=j, podemos pré-definir $y_{ij} = 0$, para evitar laços, e $c_{ij} = 1$, uma possível representação de que um nó precede ele mesmo, que não é relevante para encontrar o caminho mínimo. Ainda na estrutura c, podemos pré-definir para o nó origem o, $c_{oj} = 1$ para todo j e, analogamente, para o sumidouro s, $c_{sj} = 0$ para todo $j \neq s$, representando que a origem precede todos os outros nós e o sumidouro não precede nenhum nó, exceto ele mesmo.

As restrições envolvendo a estrutura c garantem que a solução é acíclica. Se i precede j temos c_{ij}=1 e c_{ji}=0 mas, se j também precedesse i, teríamos c_{ij}=0 e c_{ji}=1. Como não é possível satisfazer ambas valorações ao mesmo tempo, sabemos que a solução ótima não conterá ciclos. Finalmente, assim como no modelo CrossWTA, o custo do caminho deve ser minimizado.

Dois modelos SATish para o TSP baseado em fluxo foram estudados. O modelo apresentado na Listagem 3.2 utiliza a *trigger forall*. O outro utiliza a *trigger exists forall*, mostrada na listagem 3.3. Por questão de espaço, a pré-definição de valores para as estruturas y, c e w foram omitidas.

```
num=9;
y(num,num);
w(num,num);
c(num,num);

// Origem é o nó 1
intgroup flow:
forall {j,o} where j in (1,num), o in (1,1):
    not y[j][o];

// Sumidouro é o último nó
intgroup flow:
forall {k,l} where l in (1,num), k in (num,num):
    not y[k][l];

//Trigger forall
intgroup flow:
forall {j,l} where j in (1,num-1), l in (1+1,num) and j!=l:
    y[j][l];

// Fluxo unário (WTA)
intgroup wta:
forall {i,j,l} where i in (1,num-1), j in (1,num), l in (1,
  num) and j!=l:
  y[i][j] -> not y[i][l];

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

```
//Tentar conexidade - fluxo gera caminhos de mão única
intgroup int1:
forall {i,j} where i in (1,num), j in (1,num) and i!=j:
   y[i][j] -> c[i][j] and not c[j][i];

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

optgroup costo:
forall {i,j} where i in (1,num), j in (1,num):
    w[i][j] (y[i][j]) ;

penalties:
  wta level 3;
  flow level 2;
  int1 level 1;
  costo level 0;
```

Listagem 3.2: Modelo TSP baseado em fluxo - *trigger forall*

Trigger Forall

O valor mínimo da função de energia vai depender do valor da penalidade atribuído a cada nível. Isto ocorre por causa da restrição *trigger* com o repetidor *forall* e das restrições *WTA*. A restrição *trigger* associada ao repetidor *forall*, pode ser interpretada como: todos os nós devem possuir fluxo entre si. Logo em seguida, a restrição *WTA* indica que um nó deve possuir fluxo com apenas um dos outros. Assim, para cada um dos n nós, $n - 1$ restrições são violadas nesta modelagem.

A solução ótima da função representa um caminho mínimo no grafo dado. Entretanto, ao testar instâncias de maior porte (aumentando n), o valor do mínimo ficaria cada vez maior, uma vez que $n * n - 1$ restrições serão sempre violadas. Por exemplo, para uma instância onde num=6 e o caminho mínimo possui custo 20, o valor de energia ótimo é $f = 596$. Para uma instância onde num=7 e o caminho mínimo possui custo 24, o valor de energia ótimo é $f = 309949$.

A utilização da *trigger* utilizando apenas o repetidor *forall* para a modelagem do ou-exclusivo não causa problemas na modelagem do TSP. Contudo, é apontada como falha na modelagem do problema de grande porte: ARQ-PROP [1], devido a

penalização exagerada que provoca. Por isso, outro tipo de *trigger* foi estudado.

Trigger Exists

A *trigger* combinando os repetidores *exists* e *forall* foi estudada com o intuito de solucionar o problema da penalização exagerada, que ocorre com a *trigger forall*.Lembrando que o operador *exists* define que haverá pelo menos um 1 na solução encontrada, penalizando portanto a solução nula, mas não garante que haverá apenas um 1 na solução. Logo, as restrições do tipo WTA ainda se fazem necessárias. É interessante ressaltar que a ordem da utilização dos operadores influencia no resultado, isto é, a *trigger exists forall* é diferente de *forall exists*.

O modelo 3.2 permanece praticamente o mesmo, com a exceçã o da troca da *trigger forall* pela *trigger exists forall*.

```
//Trigger
intgroup flow:
exists {j} where j in (1+1,num);
forall {i} where i in (1,num-1):
    y[i][j];
```

Listagem 3.3: Trigger exists forall

O valor mínimo da função de energia equivale ao custo do caminho mínimo do grafo apresentado, representando uma vantagem sobre a modelagem anterior, tanto na interpretação do modelo quanto resolvendo o problema da penalização exagerada. Mais detalhes sobre a comparação entre os modelos utilizando a *trigger forall* e a *trigger exists forall* serão apresentados no Capítulo 4

3.3.2 TSP com nós não obrigatórios

O modelo para o TSP com nós não obrigatórios é uma extensão do modelo TSP baseado em fluxo. A utilização da *trigger forall* não é apropriada para esta extensão, uma vez que favorece as soluções com mais nós. O modelo que usa a trigger *exists forall*, que a partir de agora será citado apenas como modelo TSP baseado em fluxo, mostra-se mais apropriado para esta extensão.

O modelo utiliza a estrutura x, além das três estruturas do modelo base:

y_{ij} =1, representa o fluxo no arco ij;

c_{ij} =1, representa que o nó i precede o nó j no caminho;

w_{ij} = k, representa o custo k da viagem entre ij no caminho;

x_i =1 , representa que o nó i faz parte da solução.

As restrições que garantem que a solução ótima será acíclica, e as restrições que descrevem a função objetivo, permanecem as mesmas. Entretanto, as restrições de fluxo unário não simultâneo sofrem uma pequena alteração. Caso $x_i = 0$, o fluxo no nó i deve ser nulo, representando que o nó i não faz parte da solução.

Pré determinando valores $x_i = 1$, podemos escolher quais os nós que obrigatoriamente devem estar na solução. Os nós que não tiverem seus valores pré determinados poderão ser usados ou não na solução, conforme a conveniência para minimizar o custo do caminho.

```
num=6;
y(num,num);
w(num,num);
c(num,num);
x(num);

//Quais nós devem estar presentes
x = [1:1; 2:1; 3:1; 5:1; 6:1];

// Origem é o nó
intgroup flow:
forall {j,o} where j in (1,num), o in (1,1):
    not y[j][o];

// Sumidouro é o último nó
intgroup flow:
forall {k,l} where l in (1,num), k in (num,num):
    not y[k][l];

/*Trigger exists forall*/
intgroup flow:
exists {j} where j in (1+1,num);
forall {i} where i in (1,num-1):
    x[i] -> y[i][j];

// Fluxo unario (wtas)
intgroup wta:
forall {i,j,l} where i in (1,num-1), j in (1,num), l in (1,
   num) and j!=l:
   y[i][j] -> not y[i][l];
```

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

// Se o nó i não pertence ao subgrafo ele está isolado
intgroup flow:
forall {i,j} where i in (1,num), j in (1,num) and i!=j:
    not x[i] -> not y[i][j];

intgroup flow:
forall {i,j} where i in (1,num), j in (1,num) and i!=j:
    not x[i] -> not y[j][i];

//Tentar conexidade - fluxo gera caminhos de mão única
intgroup int1:
forall {i,j} where i in (1,num), j in (1,num) and i!=j:
    y[i][j] -> c[i][j] and not c[j][i];

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

//Minimiza o custo do caminho
optgroup costo:
forall {i,j} where i in (1,num), j in (1,num):
     w[i][j] (y[i][j]) ;

penalties:
  wta level 3;
  flow level 2;
  int1 level 1;
  costo level 0;
```

Listagem 3.4: Modelo SATish TSP com nós não obrigatórios

3.3.3 TSP com nós não obrigatórios e com custo nos nós

O modelo do TSP com nós não obrigatórios pode ser usado como base para modelar a variação que também deve minimizar custo dos nós junto com o custo das arestas. Utiliza a estrutura p, além das quatro do modelo base:

y_{ij} =1, representa o fluxo no arco ij.

c_{ij} =1, representa que o nó i precede o nó j no caminho.

w_{ij} =k, representa o custo k da viagem entre ij no caminho.

x_i =1, representa se o nó i faz parte da solução.

pi =l, representa o custo l para adicionar o nó i no caminho.

Neste caso, a restrição que define o custo a ser minimizado seria alterada para a soma dos custos das arestas incluídas na solução, com a soma do custos dos nós incluídos na solução. As restrições de fluxo unário não simultâneo, acíclico e nós não obrigatŕios permaneceriam idênticas ao modelo base.

```
p(num);
p = [1:2; 2:3; 3:4; 4:4; 5:3; 6:2]; //peso dos nós

//...

//Minimiza o custo dos nós
optgroup costo:
forall {i} where i in (1,num):
    p[i](x[i]);
```

Listagem 3.5: Modelo SATish: restrição referente ao custo dos nós

3.3.4 TSP num grafo incompleto

Até o momento, assumimos que G é um grafo completo, ou seja, o custo k da viagem entre ij é um valor finito. Contudo, podemos assumir que a estrutura y pode representar um grafo que nem ao menos possui um ciclo Hamiltoniano. O modelo TSP pode ser utilizado como base, sem nenhuma alteração na definição das estruturas e restrição. Entretanto, devemos ajustar os valores na estrutura w e y apropriadamente.

Em vez de representear o custo da viagem entre i e j, o valor w pode representar o custo da construção de uma estrada entre i e j, caso as estradas já existentes não sejam suficientes para encontrar o ciclo Hamiltoniano. Sejam

- $max(w_{ij})$ o maior custo de viagem, sabendo que a estrada existe e
- w_{kl} o custo da construção da estrada entre k e l.

Devemos definir os valores da estrutura w para que $max(w_{ij}) < w_{kl}$.

A linguagem SATish não possui uma representação de custo infinito caso existam estradas ij que não podem ser construídas sob hipótese alguma. Pode-se optar pela utilização de um valor grande, representando o infinito. Pode-se optar também pela pré-definição de $y_{ij} = 0$, ou seja, y_{ij} não será incluída na solução.

A flexibilidade da linguagem de modelagem SATish fica clara com o aproveitamento de um modelo base para um problema totalmente novo. Poderíamos modelar caminho mínimo com custo nos nós e arestas inexistentes e muitas outras variações, apenas fazendo pequenas adaptações nos modelos base.

3.4 Modelagem LogWTA

É apresentada a seguir uma nova alternativa para a representação do operador ou-exclusivo na linguagem SATish: a modelagem binária **LogWTA**. **LogWTA** visa melhorar a complexidade de espaço e diminuir o tempo de resolução. No futuro, a modelagem binária alternativa **LogWTA** pode vir a ser incorporada no próprio compilador sob a forma de um novo operador para a linguagem SATish.

Anteriormente, para modelarmos um ou-exclusivo sobre n variáveis seria necessária uma estrutura de tamanho n, e n^2 restrições do tipo WTA (modelo **CrossWTA**). A adaptação do ou-exclusivo utilizando o modelo **CrossWTA** pode trazer problemas para a representação do espaço de busca, uma vez que foi encontrada uma solução não ótima com energia menor do que a energia da solução ótima esperada para o problema de grande porte ARQ-PROP [1].

Supondo que deseja-se modelar o XOR entre n variáveis. Utilizando a modelagem **LogWTA**, constrói-se uma estrutura de dimensão $logn$ que poderá representar os valores binários entre 0 e n. O valor binário representado é equivalente a escolha única, ou seja, equivale ao operador de ou-exclusivo sobre $i_1, i_2, ..., i_n$. As restrições associadas a esta estrutura garantem a devida associação do valor binário com o restante da modelagem do problema.

LogWTA é uma das alternativas de modelagem para o problema 2-XOR, descrito em 2.4. As restrições de viabilidade são referentes a associação da modelagem binária com o problema. Não há função objetivo (restrições de otimalidade).

Para a modelagem do 2-XOR, apenas uma estrutura, *logpos*, é utilizada, de dimensão (n x logn). O algarismo menos representativo da representação binária é logpos[i][1], e a representação segue até o mais representativo logpos[i][logn]. Sendo

assim, para logn=3, por exemplo, a representação da linha i associada ao número 1 (001) seria:

not logpos[i][3] and not logpos[i][2] and logpos[i][1]

As n restrições do tipo WTA definem que não deve haver repetição, ou seja, se a linha i está associada ao número j (j está representado em formato binário), não deve haver outra linha k associada ao mesmo j. A linguagem SATish permite apenas a utilização de índices maiores ou iguais a 1, portanto a representação do número 0 (000) deve ser penalizada. Todas as restrições são de viabilidade e é utilizado apenas um nível de penalidade.

A seguir, o modelo em SATish para $n = 5$. Sabemos que $\log 5 = 3$, e com três bits é possível representar números de 0 a 7. Logo, teremos possíveis representações indesejadas para os números 6(110) e 7 (111). Estas representações devem ser penalizadas, pois não são interessantes na solução.

```
num=5;
lognum=3;
logpos(num,lognum);

//Mapeamento Representativo
//1
intgroup int1:
forall {i,j} where i in (1,num), j in (1,num) and i!=j:
    (not logpos[i][3] and not logpos[i][2] and logpos[i][1])
        ->
not (not logpos[j][3] and not logpos[j][2] and logpos[j][1])
  ;

//2
intgroup int1:
forall {i,j} where i in (1,num), j in (1,num) and i!=j:
    (not logpos[i][3] and logpos[i][2] and not logpos[i][1])
        ->
not (not logpos[j][3] and logpos[j][2] and not logpos[j][1])
  ;

//3
intgroup int1:
forall {i,j} where i in (1,num), j in (1,num) and i!=j:
    (not logpos[i][3] and logpos[i][2] and logpos[i][1]) ->
```

```
not (not logpos[j][3] and logpos[j][2] and logpos[j][1]);

//4
intgroup int1:
forall {i,j} where i in (1,num), j in (1,num) and i!=j:
    (logpos[i][3] and not logpos[i][2] and not logpos[i][1])
        ->
not (logpos[j][3] and not logpos[j][2] and not logpos[j][1])
   ;

//5
intgroup int1:
forall {i,j} where i in (1,num), j in (1,num) and i!=j:
     (logpos[i][3] and not logpos[i][2] and logpos[i][1]) ->
(not (logpos[j][3] and not logpos[j][2] and logpos[j][1]);

//Representações binárias não utilizadas (0 e valores maiores que 5)
////not 0
intgroup int1:
forall {i} where i in (1,num):
    not (not logpos[i][3] and not logpos[i][2] and not
       logpos[i][1]);

////not 6
intgroup int1:
forall {i} where i in (1,num):
    not (logpos[i][3] and logpos[i][2] and not logpos[i][1])
       ;

//not 7
intgroup int1:
forall {i} where i in (1,num):
    not (logpos[i][3] and logpos[i][2] and logpos[i][1]);

penalties:
 int1 level 0;
```

Listagem 3.6: Modelo SATish **LogWTA**

O modelo TSP baseado em fluxo possui como característica a sua flexibilidade

para modelagem da variações. Já o modelo utilizando modelagem binária **Logwta** para o TSP foi estudado com o objetivo de reduzir a complexidade de espaço.

Para a adaptação do modelo LogWTA para o TSP, a primeira dimensão da estrutura *logpos* é interpretada como a cidade e a segunda dimensão é interpretada como a posição no caminho. É acrescentada a estrutura *dist*, que armazena o custo das viagens entre as cidades i e j. A função objetivo do TSP é incorporada ao modelo base através de restrições de otimalidade do tipo:

dist[i][k]

(not logpos[i][3] and not logpos[i][2] and logpos[i][1] and

not logpos[k][3] and logpos[k][2] and not logpos[k][1]);

A restrição pode ser interpretada como: se a cidade i está na primeira posição do caminho e a cidade k está na segunda posição do caminho, então haverá uma viagem da cidade i para k. A seguir, o trecho de código que representa a função objetivo no modelo **logWTA**.

```
//1 e 2
optgroup costo:
  forall{i,j,k} where i in (1,num), k in (1,num) and i!= k:
    dist[i][k]
    (not logpos[i][3] and not logpos[i][2] and logpos[i][1]
       and
     not logpos[k][3] and logpos[k][2] and not logpos[k][1])
        ;

//2 e 3
optgroup costo:
  forall{i,j,k} where i in (1,num), k in (1,num) and i!= k:
    dist[i][k]
    (not logpos[i][3] and logpos[i][2] and not logpos[i][1]
       and
     not logpos[k][3] and logpos[k][2] and logpos[k][1]);

//3 e 4
optgroup costo:
  forall{i,j,k} where i in (1,num), k in (1,num) and i!= k:
    dist[i][k]
    (not logpos[i][3] and logpos[i][2] and logpos[i][1] and
    logpos[k][3] and not logpos[k][2] and not logpos[k][1]);
```

```
//4 e 5
optgroup costo:
  forall{i,j,k} where i in (1,num), k in (1,num) and i!= k:
    dist[i][k]
    (logpos[i][3] and not logpos[i][2] and not logpos[i][1]
       and
     logpos[k][3] and not logpos[k][2] and logpos[k][1]);
```

Listagem 3.7: Modelo SATish: Restrições de otimialidade para o TSP

Assim como no modelo CrossWTA para o TSP, pode ser necessário determinar um ciclo, ou seja, o caminho deve iniciar e terminar na cidade k. Pode-se replicar os valores da estrutura *dist* de $i = k$ para a última posição $i = n$, para todo j. Para $n = 5$, por exemplo, em que deseja-se começar e terminar na cidade 1, deve-se copiar os valores de dist[1][j] para dist[5][j] e incluir as seguintes restrições:

```
//1 é a primeira cidade do caminho
intgroup int2: not logpos[1][3];
intgroup int2: not logpos[1][2];
intgroup int2: logpos[1][1];

//5 é a última cidade do caminho (cópia de 1)
intgroup int2: logpos[5][3];
intgroup int2: not logpos[5][2];
intgroup int2: logpos[5][1];
```

Listagem 3.8: Replicação do nó 1 na posição 5

O mínimo do valor da função de energia que representa este problema é equivalente ao custo do caminho mínimo do grafo, representado no modelo.

3.5 Modelagem CrossLogWTA

Outro tipo de modelagem para o 2-XOR é a chamada **CrossLogWTA**. Não há função objetivo (restrições de otimalidade). Entretanto, diferente da modelagem **LogWTA**, as restrições de viabilidade da modelagem **CrossLogWTA** definem que as escolhas devem ser diferentes entre si, sem utilizar restrições WTA.

Utiliza-se a estrutura *logpos(2,n,logn)*. O primeiro índice representa a estrutura a ser usada (1 ou 2). O segundo índice representa os números possíveis a serem representados (1 a n). O terceiro índice possibilita a representação binária (1 a $logn$).

Assim como no modelo **LogWTA**, o algarismo menos representativo da representação binária é logpos[l][i][1], e a representação segue até o mais representativo logpos[l][i][logn]. As restrições de viabilidade associam a representação binária na estrutura 1 com o segundo índice da estrutura 2 e vice-versa. Existem portanto n^2 restrições de associação com valores binários, uma para cada possível par.

A seguir, um exemplo da restrição de viabilidade para o par 1,3, em que $logn = 3$. A estrutura 1 possui o índice do meio igual a 3 e representa o valor binário 1 (001), e a estrutura 2 possui o índice do meio igual a 1 e representa o valor binário 3 (011).

```
not logpos[1][3][3] and not logpos[1][3][2] and logpos
    [1][3][1] ->
not logpos[2][1][3] and logpos[2][1][2] and logpos[2][1][1];
```

Listagem 3.9: Restrição de viabilidade para o par (1,3) quando logn=3

Utilizando $logn = 3$, a representação binária com três posições poderia variar representar até 0 até 7. Caso $n < 7$, os valores binários não utilizados devem ser penalizados pois não são interessantes na solução. A linguagem SATish permite apenas a utilização de índices maiores ou iguais a 1, portanto a representação do número 0 (000) também deve ser penalizada. Todas as restrições são de viabilidade e é existe apenas um nível de penalidade.

```
intgroup int1:
forall {i,j} where i in (1,2), j in (1,n):
not (not logpos[i][j][3] and not logpos[i][j][2] and not
    logpos[i][j][1]);
```

Listagem 3.10: Restrição de penalização ao valor 0 (000)

A seguir, o modelo para n=7 e logn=3, que, por questão de espaço, não será apresentado completo.

```
num=7;
lognum = 3;
k = 2;
logpos(k,num,lognum);

//Mapeamento Representativo
//1,1
intgroup int1:
    not logpos[1][1][3] and not logpos[1][1][2] and logpos
        [1][1][1] ->
   (not logpos[2][1][3] and not logpos[2][1][2] and logpos
       [2][1][1]);
```

```
//1,2
intgroup int1:
    not logpos[1][2][3] and not logpos[1][2][2] and logpos
       [1][2][1] ->
   (not logpos[2][1][3] and logpos[2][1][2] and not logpos
      [2][1][1]);

//1,3
intgroup int1:
    not logpos[1][3][3] and not logpos[1][3][2] and logpos
       [1][3][1] ->
   (not logpos[2][1][3] and logpos[2][1][2] and logpos
      [2][1][1]);

//1,4
intgroup int1:
    not logpos[1][4][3] and not logpos[1][4][2] and logpos
       [1][4][1] ->
   (logpos[2][1][3] and not logpos[2][1][2] and not logpos
      [2][1][1]);

//1,5
intgroup int1:
    not logpos[1][5][3] and not logpos[1][5][2] and logpos
       [1][5][1] ->
   (logpos[2][1][3] and not logpos[2][1][2] and logpos
      [2][1][1]);

//1,6
intgroup int1:
    not logpos[1][6][3] and not logpos[1][6][2] and logpos
       [1][6][1] ->
   (logpos[2][1][3] and logpos[2][1][2] and not logpos
      [2][1][1]);

//1,7
intgroup int1:
    not logpos[1][7][3] and not logpos[1][7][2] and logpos
```

```
        [1][7][1] ->
    (logpos[2][1][3] and logpos[2][1][2] and logpos[2][1][1])
        ;

...

//7,1
intgroup int1:
    (logpos[1][1][3] and logpos[1][1][2] and logpos
        [1][1][1]) ->
    (not logpos[2][7][3] and not logpos[2][7][2] and logpos
        [2][7][1]);

//7,2
intgroup int1:
    (logpos[1][2][3] and logpos[1][2][2] and logpos
        [1][2][1]) ->
    (not logpos[2][7][3] and logpos[2][7][2] and not logpos
        [2][7][1]);

//7,3
intgroup int1:
    (logpos[1][3][3] and logpos[1][3][2] and logpos
        [1][3][1]) ->
    (not logpos[2][7][3] and logpos[2][7][2] and logpos
        [2][7][1]);

//7,4
intgroup int1:
    (logpos[1][4][3] and logpos[1][4][2] and logpos
        [1][4][1]) ->
    (logpos[2][7][3] and not logpos[2][7][2] and not logpos
        [2][7][1]);

//7,5
intgroup int1:
    (logpos[1][5][3] and logpos[1][5][2] and logpos
        [1][5][1]) ->
    (logpos[2][7][3] and not logpos[2][7][2] and logpos
```

```
        [2][7][1]);

//7,6
intgroup int1:
    (logpos[1][6][3] and logpos[1][6][2] and logpos
        [1][6][1]) ->
    (logpos[2][7][3] and logpos[2][7][2] and not logpos
        [2][7][1]);

//7,7
intgroup int1:
    (logpos[1][7][3] and logpos[1][7][2] and logpos
        [1][7][1]) ->
    (logpos[2][7][3] and logpos[2][7][2] and logpos
        [2][7][1]);


//Representações binárias não utilizadas
//(not 0)
intgroup int1:
forall {i,j} where i in (1,k), j in (1,num):
   not (not logpos[i][j][1] and not logpos[i][j][2] and not
       logpos[i][j][3]);

penalties:
 int1 level 0;
```

Listagem 3.11: Modelo SATish para **CrossLogWTA**

Por trabalhar com duas estruturas, este modelo possui a característica de encontrar pares espelhados na solução ótima. Ou seja, se a estrutura 1 encontrar na posição 5 o valor binário 011, referente ao número 3, a estrutura 2 possuirá na posição 3 o valor binário 101, referente ao número 5.

3.6 Modelagem TreeWTA

TreeWTA é mais uma alternativa que usa variáveis binárias para a modelagem do 2-XOR, problema descrito em 2.4. O objetivo é construir uma árvore binária, onde a raíz representa todas as escolhas de 1 a n, os dois nós do primeiro nível representam a seleção de 1 a $n/2$ ou $(n/2)+1$ a n, e a divisão segue até o nível das folhas, onde

há um nó para cada uma das n escolhas. As restrições garantem que apenas uma folha dentre as n possíveis, seja selecionada. Para o 2-XOR, realiza-se n escolhas, portanto haverá uma árvore para cada escolha: n árvores. Para garantir que as escolhas sejam diferentes entre si, utilizamos restrições WTA convencionais.

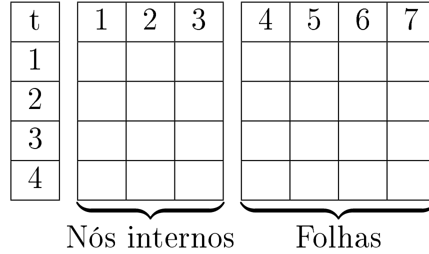

Figura 3.1: Representação do modelo TreeWTA para $n = 4$

Utiliza-se a estrutura $p(n, num * 2 - 1)$, onde a primeira dimensão representa o índice da árvore e a segunda dimensão representa a árvore em si. A representação da árvore é tal que o nó i é pai de $i * 2$ e $(i * 2) + 1$. Assim, os índices 1 a $n - 1$ representam os nós internos e os índices n a $umn - 1$ representam as folhas, como mostrado na figura 3.1.

```
num=12;
p(num,(num*2)-1);    //n pai de n*2 e (n*2)+1

p = [1,1:1; 2,1:1; 3,1:1; 4,1:1; 5,1:1; 6,1:1; 7,1:1; 8,1:1;
     9,1:1; 10,1:1; 11,1:1; 12,1:1];

//Árvore
intgroup int1:
forall {i,j} where i in (1,num), j in (1,num-1):
     p[i][j] -> p[i][j*2] or p[i][(j*2)+1];

intgroup int1:
forall {i,j} where i in (1,num), j in (1,num-1):
     p[i][j*2] or p[i][(j*2)+1] ->  p[i][j];

//WTA de nós vizinhos
intgroup int1:
forall {i,j} where i in (1,num), j in (1,num-1):
     p[i][j*2] -> not p[i][(j*2)+1];

intgroup int1:
forall {i,j} where i in (1,num), j in (1,num-1):
```

```
    p[i][(j*2)+1] -> not p[i][j*2];

//WTA 2a dimensão
intgroup int1:
forall {i,j} where i in (1,num), j in (1,num), k in (num,(
  num*2)-1) and i!=j:
    p[i][k] -> not p[j][k];

penalties:
   int1 level 0;
```

Listagem 3.12: Modelo SATish para **TreeWTA**

Para a interpretação dos resultados, deve-se pegar o índice j equivalente a folha selecionada subtrair $(num - 1)$. Por exemplo, para $num = 4$, se a linha 2 escolher o ítem 3, teremos o valor 1 nos nós internos t_{21}, t_{23} e na folha t_{26}, e as outras desta linha terão o valor 0. t_{21} representa a raíz da árvore 2, e a folha t_{26} representa o ítem 3, já que $6 - (4 - 1) = 3$.

Uma vantagem desta modelagem é a possibilidade da utilização de uma notação compacta na linguagem SATish. A adaptação do modelo para instâncias maiores requer apenas a alteração do valor da constante *num*.

3.7 Modelagem CrossTreeWTA

Para modelar o 2-XOR, descrito em 2.4, a abordagem CrossTreeWTA utiliza duas árvores binárias, uma em cada dimensão do problema. Não utiliza restrições WTA, exceto para a modelagem da árvore em si. A árvore t representa a primeira dimensão, e a árvore c representa a segunda dimensão. A estrutura c apresenta apenas os nós internos, uma vez que compartilha as folhas com a estrutura t. A representação da árvore c está na vertical, enquanto a representação da árvore t está na horizontal. As folhas da estrutura t são as linhas de t enquanto as folhas de c são as colunas de t.

Na Figura 3.2 temos uma representação da relação entre as estruturas t e c, para $num{=}4$. Para $num = 4$, temos que c_{21} é pai de t_{14} e t_{24} e c_{31} é pai de t_{34} e t_{44}.

O modelo **CrossTreeWTA** é baseado no modelo **TreeWTA**. A modelagem da árvore binária t, que representa a primeira dimensão, é igual para ambos os modelos. Entretanto, as restrições WTA da modelagem **TreeWTA** são substituídas pela segunda árvore binária, mostrada na Listagem 3.13.

Restrições para nós intermediários da segunda dimensão são semelhantes aos da primeira dimensão, exceto para os nós do penúltimo nível. Os nós do penúltimo nível

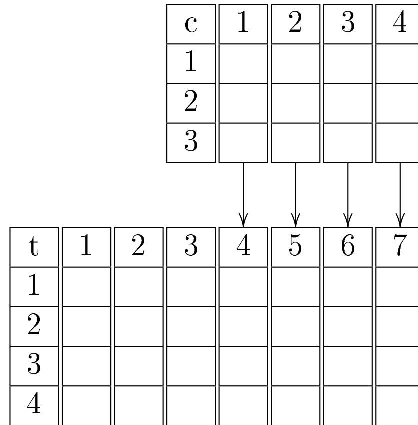

Figura 3.2: Representação do modelo CrossTreeWTA

de c serão pais das folhas, e as folhas estão nas colunas estrutura t. Naturalmente, as restrições WTA para nós vizinhos também é utilizada para as folhas da árvore da segunda dimensão.

```
//2a dimensão
//Arvore: Nos intermediários
intgroup int1:
forall {i,j} where i in (1,(num−1)/2), j in (1,num):
   c[i][j] −> c[i*2][j] or c[(i*2)+1][j];

intgroup int1:
forall {i,j} where i in (1,(num−1)/2), j in (1,num):
    c[i*2][j] or c[(i*2)+1][j] −>  c[i][j];

//Arvore: Folhas - n PAR
intgroup int1:
forall {i,j,k} where i in (num/2,num−1), j in (1,num), k in
   (num,num):
    c[i][j] −> t[(1+i−k/2)*2][j+k−1] or t[(1+i−k/2)*2 −1][j+k
        −1];

intgroup int1:
forall {i,j} where i in (num/2,num−1), j in (1,num), k in (
   num,num):
   t[(1+i−k/2)*2][j+k−1] or t[(1+i−k/2)*2 −1][j+k−1] −>  c[i
        ][j];

//WTA de nós vizinhos
intgroup int1:
```

```
forall {i,j} where i in (1,num/2), j in (1,num), k in (num,
    num):
      t[i*2-1][j+k-1] -> not t[i*2][j+k-1];

intgroup int1:
forall {i,j} where i in (1,num/2), j in (1,num), k in (num,
    num):
    t[i*2][j+k-1] -> not t[i*2-1][j+k-1];
```

Listagem 3.13: Restrições SATish adicionais para **CrossTreeWTA**

Para o caso de n ser ímpar, é necessária uma adaptação no modelo, pois os nós na última linha da estrutura c terão apenas um filho na estrutura t, e não dois. A última linha c_{ij} onde $j = num - 1$ deve ser tratada com uma restrição diferente, como mostrado na Listagem 3.14, para não gerar um índice fora dos limites.

```
//Arvore: Folhas - n ímpar
intgroup int1:
forall {i,j,k} where i in (num/2,num-2), j in (1,num), k in
    (num,num):
      c[i][j] -> t[(1+i-k/2)*2][j+k-1] or t[(1+i-k/2)*2-1][j+k
         -1];

intgroup int1:
forall {i,j} where i in (num/2,num-2), j in (1,num), k in (
    num,num):
    t[(1+i-k/2)*2][j+k-1] or t[(1+i-k/2)*2-1][j+k-1] ->  c[i
       ][j];

//Arvore: Folhas - filho único
intgroup int1:
forall {i,j,k} where i in (num-1,num-1), j in (1,num), k in
    (num,num):
      c[i][j] -> t[(1+i-k/2)*2-1][j+k-1];

intgroup int1:
forall {i,j} where i in (num-1,num-1), j in (1,num), k in (
    num,num):
      t[(1+i-k/2)*2-1][j+k-1] ->  c[i][j];
```

Listagem 3.14: **CrossTreeWTA** quando n é ímpar

Capítulo 4

Resultados e Discussão

Todos os testes descritos neste capítulo foram feitos em máquinas com uma das seguintes configurações 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 2.6.38-11
- Processador 0: Intel(R) Pentium(R) Dual CPU T2390 @3.00GHz
- Processador 1: Intel(R) Pentium(R) Dual CPU T2390 @3.00GHz
- Memória: 2GB
- Sistema Operacional: GNU/Linux Kernel 2.6.38-11

O resolvedor Bonmin [14] é chamado a partir da integração do SATyrus com AMPL. A versão do Bonmin utilizada para testes foi a 1.5.1. Nos testes realizados, os parâmetros resolve_ at_ node e resolve_ at_ root (mencionados neste capítulo como node e root, respectivamente) foram adicionandos ao arquivo gerado pelo SATyrus na linguagem AMPL, para tentar melhorar a qualidade das soluções encontradas. Ambos são parâmetros de números inteiros positivos e tem como padrão o valor 0.

O ajuste de parâmetros foi baseado em potências de dois. Este método foi escolhido porque pequenas variações na configuração, como de um em um por exemplo, não provocam diferenças significativas na resolução e gerariam uma quantidade muito grande de possibilidades a serem testadas. Diversas configurações possíveis podem levar a obtenção da solução ótima. Neste caso, a escolhida foi a que levou menor tempo. A variação de tempo entre execuções, dados os mesmos parâmetros,

é pequena em relação ao tempo total de execução. Após a seleção da melhor configuração para os parâmetros do Bonmin, realizou-se 5 execuções para a obtenção de uma média do tempo de execução.

```
model tsp10.mod;
option solver bonmin;
options bonmin_options "bonmin.num_resolve_at_node 4";
solve;
display {i in 1.._nvars} (_varname[i], _var[i]);
```

Listagem 4.1: Exemplo do arquivo gerado em AMPL pelo SATyrus, acrescentando o parâmetro do resolvedor

As modelagens **CrossWTA**, **LogWTA**, **CrossLogWTA**, **TreeWTA** e **CrossTreeWTA** para o problema 2-XOR foram testadas com instâncias de 7 as 17 nós. As modelagens para o TSP foram testadas com uma instância que possui apenas uma solução ótima, na qual o caminho mínimo é em ordem crescente, para facilitar a visualização dos resultados. O custo no caminho ótimo é $(n-1)*4$. Foram utilizadas instâncias de 6 a 9 nós.

Tabela 4.1: Exemplo de custos (simétricos) de uma instância do TSP com 6 cidades (nós)

	1	2	3	4	5	6
1	0	4	20	20	20	20
2	4	0	4	20	20	20
3	20	4	0	4	20	20
4	20	20	4	0	4	20
5	20	20	20	4	0	4
6	20	20	20	20	4	0

4.1 CrossWTA

Na Tabela 4.2, são apresentados resultados detalhados dos testes com o modelo **CrossWTA**. A diferença no valor da energia mínima, entre as instâncias de um mesmo modelo, ocorre porque o mínimo depende de n, devido a utilização da *trigger forall*, como visto no Capítulo 3. A energia mínima da abordagem CrossWTA para o 2-XOR é $n*(n-1)$ porque, por utilizar a *trigger forall*, o modelo atribui a penalidade 1, os $n-1$ ítens não escolhidos para parear com cada um dos n ítens.

É também possível observar uma diferença maior do que o custo do caminho no valor da energia mínima entre instâncias de mesmo tamanho para o modelo 2-XOR e o modelo TSP. Isso ocorre porque o modelo 2-XOR utiliza apenas um

Tabela 4.2: Resultados para o modelo CrossWTA

Modelo	Nós	Tempo	Parâmetro (root/node)	Energia
2-XOR	7	0,37	0/0	42
	8	0,47	0/0	56
	9	0,86	0/0	72
	10	0,78	2/0	90
	11	0,86	2/0	110
	12	1,06	2/2	132
	13	1,92	2/0	156
	14	2,85	2/2	182
	15	3,21	2/2	210
	16	3,92	2/0	240
	17	6,25	2/0	272
TSP	6	0,11	-	9040
	7	0,21	-	21216
	8	0,33	-	43960
	9	0,48	-	83008

nível de penalidade enquanto o segundo utiliza dois níveis, um para a restrição de otimalidade e outro para as restrições de viabilidade. Ou seja, as restrições de viabilidade violadas pela *trigger forall* no problema do TSP somam uma penalidade maior do que no modelo 2-XOR.

4.2 TSP baseado em fluxo

No modelo para o TSP baseado em fluxo, a interpreteção da solução é direta, uma vez que o valor um na posição ij representa que a aresta ij está no subgrafo solução. Na tabela 4.3, encontra-se a representação dos valores da estrutura y na solução ótima, de custo 20 (caminho 1 2 3 4 5 6), encontrada para a instância de 6 nós do TSP 4.1. A estrutura c é intermediária, utilizada apenas para detecção de ciclos, está representada na tabela 4.4.

Tabela 4.3: Exemplo de solução ótima do TSP baseado em fluxo

y	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	0	1	0	0	0
3	0	0	0	1	0	0
4	0	0	0	0	1	0
5	0	0	0	0	0	1
6	0	0	0	0	0	0

Tabela 4.4: Exemplos de valores da matriz c numa solução ótima do TSP baseado em fluxo

c	1	2	3	4	5	6
1	1	1	1	1	1	1
2	0	1	1	1	1	1
3	0	0	1	1	1	1
4	0	0	0	1	1	1
5	0	0	0	0	1	1
6	0	0	0	0	0	1

4.2.1 *Trigger forall*

Neste modelo, a energia obtida na solução não corresponde diretamente ao custo mínimo do caminho, como visto em 3.3.1. Entretanto, o custo pode ser calculado subtraindo da energia obtida, a penalidade de cada uma das $n-2*n-2$ restrições violadas até obter o custo do caminho.

Por exemplo, para uma instância com 6 nós, a energia da solução ótima é 596. As restrições violadas estão associadas ao nível de penalidade 1. O compilador atribuiu o valor 36 para a penalização deste nível. Sabemos que 16 restrições foram violadas, portanto 576 equiavale a penalizações. Subtraindo a energia obtida com o valor das penalizações, obtemos o custo do caminho mínimo, que é 20. Na Tabela, as 16 arestas que são penalizados na solução encontrada estão marcadas com x, as arestas que fazem parte da solução, com o, e as arestas descartadas pelas restrições, com -.

Tabela 4.5: Arestas penalizadas na solução ótima do TSP baseado em fluxo com *trigger forall*

Destino/Origem	1	2	3	4	5	6
1	-	-	-	-	-	-
2	o	-	x	x	x	-
3	x	o	-	x	x	-
4	x	x	o	-	x	-
5	x	x	x	o	-	-
6	x	x	x	x	o	-

Por ser dependente do valor de n, o mínimo é um valor cada vez maior, e o custo efetivo do caminho torna-se pequeno em relação ao valor da função de energia, podendo dificultar a busca. Portanto, o modelo mostra-se pouco escalável. Para uma instância com 8 nós, a busca pelo mínimo já leva por volta de quarenta minutos. A solução ótima possui custo 28 e energia f=905500, como detalhado na tabela 4.6.

Tabela 4.6: Resultados para o modelo TSP baseado em fluxo - *trigger forall*

Nós	Tempo	Parâmetro (root/node)	Energia
6	845,77	512/64	596
7	516,55	0/128	309.949
8	8735,81	16384/8192	905500
9	15079,32	16384/4096	2290145

4.2.2 *Trigger exists forall*

Este modelo possui uma vantagem sobre o anterior: o valor da função de energia representa exatamente o custo do caminho mínimo, facilitando a interpretação da solução.

Tabela 4.7: Resultados para o modelo TSP baseado em fluxo - *trigger exists*

Nós	Tempo	Parâmetro (root/node)	Energia
6	12,19	-	20
7	143,03	0/128	24
8	1006,11	256/2048	28
9	13431,71	16384/32768	32

Os tempos de resolução para o modelo que usa a *trigger exists* foram menores. Entretanto, para a instância com 9 nós, os tempos foram bem próximos.

4.3 Modelagem LogWTA

Na Tabela 4.9 são apresentados resultados detalhados dos teste com o modelo LogWTA. A modelagem para o problema 2-XOR não possui restrição de otimalidade, portanto a energia mínima esperada é 0. Sua adaptação para o TSP possui como mínimo o custo exato do menor caminho, facilitando a interpretação dos resultados.

Instâncias maiores do que 16 nós implicam em aumentar a representação binária de 4 posições para 5. Mesmo com diversas tentativas de configurações de parâmetros, não foi encontrada uma que levasse ao ótimo. A obtenção do ótimo, neste caso, deve ser fruto de estudos futuros.

4.4 Modelagem Crossed LogWTA

A abordagem **CrossedLogWTA** possui a desvantagem de necessitar de duas estruturas.

É possível observar na Tabela 4.11 que os testes mostraram resultados de tempo maior do que as outras aboradgens. Assim como a abordagem LogWTA, a instân-

Tabela 4.8: Exemplo de resultado para a abordagem *LogWTA*

	Binário			
Índice	3	2	1	Representação
1	0	0	1	1
2	0	1	0	2
3	0	1	1	3
4	1	0	0	4
5	1	0	1	5
6	1	1	0	6

Tabela 4.9: Tempo em segundos para instâncias da abordagem LogWTA

Modelo	Nós	Tempo(s)	Parâmetro(root/node)	Energia
2-XOR	7	0.41	0/0	0
	8	1.03	0/0	0
	9	1.93	2/0	0
	10	4.82	0/4	0
	11	3.65	0/2	0
	12	5.44	2/0	0
	13	9.58	4/0	0
	14	2.01	2/0	0
	15	4.98	4/0	0
TSP	6	0,57	0/0	20
	7	11,53	0/0	24
	8	1886,67	0/0	28
	9	649,15	0/0	32

cia de 16 nós implica em acrescentar um bit na representação binária. Não foi encontrada uma configuração de parâmetros que levasse ao ótimo, neste caso.

4.5 Modelagem TreeWTA

Este modelo possui a vantagem de facilitar de adaptação para instâncias maiores requer apenas a alteração do valor da constante *num*. Uma desvantagem em relação as abordagens **LogWTA** e **CrossLogWTA** é a complexidade de espaço, pois a estrutura utilizada é de dimensão $2n^2$.

4.6 Modelagem CrossTreeWTA

Não foram encontradas configurações de parâmetros que levassem instâncias maiores do que 7 a solução ótima.

Tabela 4.10: Exemplo de resultado para a abordagem Cross LogWTA

Logpos[1][i][j]	1	2	3	Número
1	1	0	1	5
2	1	0	0	4
3	1	1	0	6
4	0	0	1	1
5	0	1	1	3
6	0	1	0	2
7	1	1	1	7

Tabela 4.11: Tempo em segundos para instâncias da abordagem CrossLogWTA

Nós	Tempo(s)	Parâmetro(root/node)
7	0,39	2/0
8	133,2	0/64
9	48,6	2/8
10	176,2	16/128
11	892,0	8/512
12	3841,25	32/1024
13	2904,36	32/1024
14	3328,07	32/1024
15	9855,2	128/1024

Tabela 4.12: Tempo em segundos para instâncias da abordagem TreeWTA

Nós	Tempo	Parâmetro (root/node)
7	2,26	0/16
8	1,36	0/8
9	48	0/64
10	185	0/256
11	2287	0/2048
12	1246	0/512

Capítulo 5

Conclusões

5.1 Sumário

O problema 2-XOR foi estudado como subproblema para a modelagem do TSP na linguagem SATish. As abordagens foram **CrossWTA**, **LogWTA**, **CrossLogWTA**,**TreeWTA** e **CrossTreeWTA**. Na Tabela 5.1 é possível observar a comparação entre os modelos quanto ao número de variáveis e restrições. Na Tabela 5.2 podemos observar a comparação quanto ao tempo de resolução dos modelos pelo resolvedor Bonmin.

Nem o modelo com menor quantidade de variáveis **LogWTA**, nem o modelo com menor quantidade de restrições **CrossLogWTA** alcançaram os menores tempos de resolução. O modelo com menores tempos de resolução encontrados foi o **CrossWTA**. O modelo **LogWTA** ficou em segundo lugar, entretanto não foi possível comparar as instâncias de 16 e 17 nós pois não foi encontrada uma configuração dos parâmetros que levasse ao mínimo. A mesma dificuldade ocorreu para o modelo **CrossTreeWTA**. Não foi possível comparar instâncias maiores do que 13 para o modelo **TreeWTA** devido a uma limitação da versão estudante do sofware AMPL, que não permite mais do que 300 variáveis no problema.

Tabela 5.1: Comparação das modelagens do problema 2-XOR

Abordagem	*Variáveis*	*Restrições*
CrossWTA	n^2	$n^3 + n^2$
LogWTA	$n \log n$	$n^3 + n * (2^{\lfloor (\log n)+1 \rfloor} - n)$
CrossLogWTA	$2n \log n$	$n^2 + 2n * (2^{\lfloor (\log n)+1 \rfloor} - n)$
TreeWTA	$2n^2 - n$	$(3n^3 + 5n^2 - 6n)/2$
CrossTreeWTA	$3n^2 - 2n$	$7n^2$

O modelo **CrossWTA** também alcançou os menores tempos nas instâncias do TSP, conforme visto na Tabela 5.4. Dentre os modelos de fluxo, o modelo com a *trigger exists* obteve melhores resultados do que o modelo com a *trigger forall*, além

Tabela 5.2: Comparação do tempo de resolucao para diferentes modelagens do problema 2-XOR

Método/Nós	7	8	9	10	11	12	13	14	15	16	17
CrossWTA	0,37	0,47	0,86	0,78	0,86	1,06	1,92	2,85	3,21	3,92	6,25
LogWTA	0,41	1,03	1,93	4,82	3,65	5,44	9,58	2,01	4,98	-	-
CrossLogWTA	0,39	133	48	176	896	3841	2904	3328	9855	-	-
TreeWTA	2,26	1,36	48,83	185	2287	1246	-	-	-	-	-

da vantagem de o mínimo da energia representar exatamente o custo do caminho mínimo.

Como mostrado no capítulo 3, o modelo TSP baseado em fluxo possui uma modelagem flexível, permitindo a sua adaptação para variações como considerar o grafo incompleto, custo nos nós e nós não obrigatórios. Entretanto, a flexibilidade gera um custo adicional no tempo de resolução que aumenta com o tamanho da instância. O tempo é dez vezes maior para 6 nós, e já mil vezes maior para 8 nós, em comparação ao melhor resultado **CrossWTA**.

Tabela 5.3: Comparação das modelagens do problema TSP

Abordagem	*Variáveis*	*Restrições*
CrossWTA	n^2	$n^3 - n^2$
Fluxo	$2n^2$	$3n^3 + 3n^2 + 2n$
LogWTA	$2n \log n$	$n^2 + 2n * (2^{\lfloor (\log n)+1 \rfloor} - n)$

Tabela 5.4: Comparação do tempo de resolução para diferentes modelagens do problema TSP

Método / Nós	6	7	8	9
CrossWTA	0,11	0,21	0,33	0,48
Fluxo trigger forall	845,77	516,55	8735,81	15079,32
Fluxo trigger exists	12,19	143,03	1023,44	2686,34
LogWTA	0,57	11,53	1886,67	649,15

5.2 Trabalhos Futuros

Parâmetros para o resolvedor

Os experimentos foram realizados apenas com parâmetros potências de dois. Uma exploração melhor dos possíveis parâmetros pode-se encontrar o mínimo em menor tempo e até mesmo alcançar o mínimo onde não foi possível neste trabalho. Além disso, Bonmin possui diversos parâmetros para personalizar o algoritmo BB para adequar-se a um problema específico. Apenas resolve_ at_ node e resolve_ at_-root foram explorados neste trabalho. Outros como allowable_ gap, allowable_ frac-

tion_ gap e cutoff_ decr também podem ser explorados, inclusive em combinação com os já citados, de forma a obter uma resolução mais eficiente.

Novos operadores para a linguagem SATish

Para facilitar ainda mais a utilização da linguagem SATish, é possível incorporar estratégias de resolução do ou-exclusivo ao compilador do SATyrus sob a forma dos operadores exists_one e/ou exor na linguagem SATish.

Eliminar a dependência da linguagem AMPL

O resolvedor Bonmin é chamado a partir de uma modelagem da função de energia gerada pelo ambiente SATyrus na linguagem AMPL. Esta dependência prejudica os experimentos pois AMPL é não é um software livre, e sua versão estudante possui a limitação de problemas até trezentas variáveis. A criação de uma nova maneira para invocar Bonmin eliminaria estas dificuldades.

Desenvolvimento de um resolvedor específico

A função de energia gerada pelo mapeamento tem como característica, apenas termos de primeiro grau. O desenvolvimento de um resolvdor específico para este tipo de função pode melhorar os resultados encontrados pelo ambiente SATyrus.

Melhorias na linguagem

Uma das dificuldades para os testes com instâncias de maior porte para o TSP foi a atribuição individual de custos para cada elemento da estrutura. Por exemplo, para a estrutura com 15 nós testada, foi necessário atribuir 225 valores de custos, um por um. Uma possível melhoria para o compilador SATyrus seria a criação de um repetidor com a definição de índices, assim como nas restrições, para atribuir valores iguais dentro de uma estrutura.

Para estruturas binárias a facilidade pode ser ainda maior, criando-se um operador que apenas cite os índices ou intervalos de índices dos valores pré definidos em 1 e um outro operador que cite os valores pré-definidos em 0. Por exemplo:

setone x[1-3; 6]; setzero x[4; 7];

A própria definição de índices pode ser aprimorada na linguagem para permitir a definição de intervalos disjuntos. (1-4;7-9;13-15) por exemplo.

Referências Bibliográficas

[1] MONTEIRO, B. F. *SATYRUS2: Compilando especificações de raciocínio lógico*. Tese de Mestrado, Universidade Federal do Rio de Janeiro, março 2010.

[2] ESPINOZA, M. M. M. M. *Compilando Resolução de Problemas para Minimização de Energia*. Tese de Mestrado, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, setembro 2006.

[3] HOPFIELD, J. "Neural Networks and Physical Systems with Emergent Collective Computational Abilities". In: *In Proceedings of the National Academy of Sciences*, pp. 2554–2558, USA, 1982.

[4] KIRKPATRICK, S., GELLAT, C. D., VECCHI, M. P. "Optimization by Simulated Annealing". In: *In Science, New Series, VOL. 220, No. 4598*, 1983.

[5] HINTON, G., SEJNOWSKI, T., ACKLEY, D. "Boltzman Machines: Constraint Satisfaction Networks that Learn". In: *In Technical Report CMU-CS84 119*, 1984.

[6] PINKAS, G. *Logical Inference in Symmetric Neural Networks*. Tese de Doutorado, Sever Institute of Technology, 1992.

[7] LIMA, P. M. V., MORVELI-ESPINOZA, M. M., FRANÇA, F. M. G. "Logic as Energy: A SAT-Based Approach". In: *BVAI*, v. LNCS, pp. 458–467, 2007.

[8] LIMA, P. M. V. "A Goal-Driven Neural Propositional Interpreter", *Int. J. Neural Syst.*, v. 11, n. 3, pp. 311–322, 2001.

[9] 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.

[10] LISSER, A., PLATEU, G., MACULAN, N. "Integer linear models with a polynomial number of variables and constraints for some classical combinatorial optimization problems", *Pesquisa Operacional*, v. 23, n. 1, pp. 161–168, 2003. ISSN: 0101-7438. `http://dx.doi.org/10.1590/S0101-74382003000100012`.

[11] HOPFIELD, J., TANK, D. W. "Neural Computation of Decisions in Optimization Problem". In: *Biological Cybernetics, 52*, pp. 141–152, 1984.

[12] PINKAS, G., LIMA, P. M. V., COHEN, S. "Compact Crossbars of Multi-Purpose Binders for Neuro-Symbolic Computation". pp. 14–18, Barcelona, Catalonia, Spain, 2011. NeSy 11 - Seventh International Workshop on Neural-Symbolic Learning and Reasoning.

[13] NOVELLO, C. F., COHEN, S., MACULAN, N., FRANÇA, F. M. G., PINKAS, G., LIMA, P. M. V. "XOR as MILP alternative modelings". pp. 105–108. Global Optimization Workshop, 2012.

[14] 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`>.

[15] PEREIRA, G. *Mapeamento e combinação de problemas NP-difíceis através de restrições pseudo-booleanas para redes neuronais articiais*. Tese de Mestrado, Universidade Federal do Rio de Janeiro, setembro 2006.

[16] DIACOVO, R. *Resolvendo o problema de planejamento da expansão da geração de energia com enxames de partículas binários*. Tese de Mestrado, Universidade Federal do Rio de Janeiro, março 2008.

[17] 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.

[18] BOOLOS, G. S., BURGESS, J. P., JEFFREY, R. C. *Computability and Logic*. 4 ed. Cambridge, Cambridge University Press, 2006.

[19] SZWARCFITER, J. L. *Grafos e Algoritmos Computacionais*. 2 ed. Rio de Janeiro, Editora Campus, 1986.

[20] NETTO, P. O. B. *Grafos: Teoria, Modelos, Algoritmos*. 4 ed. São Paulo, Editora Edgard Blucher, 2006.

[21] LIMA, P. M. V., MORVELI-ESPINOZA, M. M., PEREIRA, G. C., FRANCA, F. M. G. "SATyrus: A SAT-based Neuro-Symbolic Architecture for Constraint Processing", *Hybrid Intelligent Systems, International Conference on*, v. 0, pp. 137–142, 2005.

[22] LIMA, P. M. V., ESPINOZA, M. M. M., PEREIRA, G. C., FERREIRA, T. D. O., FRANÇA, F. M. G. "Logical Reasoning via Satisfiability Mapped into Energy Functions", *International Journal of Pattern Recognition and Artificial Intelligence*, v. 22, pp. 1031–1043, 2008.

[23] GEMAN, S., GEMAN, D. "Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images", pp. 564–584, 1987.

[24] *AMPL Modeling Language for Mathematical Programming.* Disponível em: <`http://www.ampl.com/`>.

[25] *FICO™ Xpress Optimization Suite 7.* Disponível em: <`http://www.fico.com/en/Products/DMTools/Pages/FICO-Xpress-Optimization-Suite.aspx`>.

[26] BONAMI, P., KILINC, M., LINDEROTH, J. *Algorithms and Softare for Convex Mixed Integer NonLinear Programs.* In: Report 1664, University of Wisconsin-Maddison, 2009.

[27] LIMA, P. M. V., PEREIRA, G. C., ESPINOZA, M. M. M., FRANÇA, F. M. G. "Mapping and Combining Combinatorial Problems into Energy Landscapes via Pseudo-Boolean Constraints". In: *BVAI*, v. LNCS, pp. 308–317, 2005.