Modelos Computacionais com Programação por Restrições

Claudio Cesar de Sá e outros

15 de Março de 2016

Conteúdo

Nota desta Versão Experimental

Autores

Este é um livro experimental (projeto de uma publicação a longo prazo) e tem muitos autores. Dentre eles cita-se:

1. Marcos Creuz Filho
2. Lucas Hermman Negri
3. .....
4. Ajudando ... reservado para incluirmos o seu nome ...

Sobre o Livro

1. Este livro tem um foco: é o título do livro. Alguns modelos clássicos de problemas, alguns originais, são discutidos, modelados e implementados em Minizinc.
2. O Minizinc foi escolhido pela sua leitura próxima a formulação matemática. Voce irá gostar desta linguagem orientada a modelos;
3. Este livro, parte dele, é resultado de alguns semestres de ensino de graduação no curso de Ciência da Computaçao da UDESC, de uma disciplina optativa introdutória a Programação por Restrições (PR).

Resumindo ....

A proposta deste livro é completamente **embrionária**. Neste primeiro momento, há um *copy-paste* de um TCC aqui da UDESC–Joinville, que orientei. Contudo, há muito material escrito que falta organizar dentro desta proposta.

Assim, vá me cobrando por email, `claudio.sa@udesc.br`, que vou aprontando o material na medida que as perguntas surgerirem.

1

Complexidade de Problemas

Neste capítulo são apresentados os fundamentos sobre complexidade de problemas. Avaliar o que são problemas exponenciais e o quanto são complexos.

1.1 Mapeando o Território

Os problemas combinatoriais são ubíquos no cotidiano de pessoas, indústria, etc. Basicamente utiliza-se alguma estimativa entre combinar objetos, pessoas versus recursos para se atingir algum objetivo. Em termos matemáticos, tem-se variáveis tais como x_1, x_2, x_3, etc, a terem seus valores instanciados sobre algum domínio ou terem alguma relação com outras variáveis, segundo algum critério ou requisito.

Exemplificando, voce quer organizar um calendário de jogos para os times da sétima divisão de sua cidade. Assim, os n times devem fazer um rodízio nos 2 campos disponíveis, tal que todo sábado tenham 2 jogos por campo. Logo, serão 8 times jogando num final de semana. Basicamente voce deve distribuir estes encontros entre dois times respeitando critérios como:

1. Local do jogo: todos os times devem jogar igualmente nos dois campos. Afinal, um deles tem grama e a bola *corre redonda*;

2. Espaçamento entre os encontros: os times devem jogar até o final da temporada;

3. Devem alternar o horário de jogo: afinal, se seu time tiver o primeiro jogo da tarde, a feijoada está comprometida.

Enfim, este simples ensaio mostra o cotidiano de situações combinatoriais em que tudo se apresenta. Estes problemas vem sendo atacados há séculos tendo Euler como um dos matemáticos precursores, ao analisar o problema das 7 pontes das Königsberg. A representação por meio de grafos foi uma estratégia de analisar e solucionar o problema. O problema do caminho euleriano é base de vários estudos, pois sua tratabilidade o coloca na classe de problemas NP [17].

Neste encaminhamento, várias áreas de pesquisa estão focadas como: Pesquisa Operacional, Computação Evolutiva, Inteligência Artificial (IA), Programação por Restrições (PR), etc. O contexto desta pesquisa encontra-se dentro destas duas últimas áreas, mais especificamente a Programação por Restrições.

1.1.1 Problemas em Inteligência Artificial

Diversas áreas estão associadas à Inteligência Artificial como tradução de línguas, interpretação de regras, robótica em geral, manipulação de dados em imagens, planejamento, reconhecimento e outras atividades. Uma das caraterísticas destes problemas é a sua complexidade algoritmica e computacional, que em geral crescem exponencialmente. Dado este fato, há uma dificuldade inerente na tarefa do desenvolvimento de algoritmos e solucoes aceitáveis. Estes aspectos se relacionam a área de tratabilidade de problemas, os quais delineam as fronteiras dos limites dos problemas em aceitarem soluções em um tempo razoáveis de processamento.

Para encontrar uma solução aos problemas abordados pela IA é necessário que o processo de análise seja detalhado e modelado de acordo com técnicas específicas. Neste sentido duas áreas da IA são destacadas [16]: *representação do conhecimento* e aplicação de *métodos de busca*. Estas duas áreas apresentam uma interseção a Programação por Restriçõespois os problemas quando atacados com PR necessitam apresentar um *modelo* a ser computado e esquemas de buscas sobre os espaço de estados que o mesmo exibe. Um dos resultados de uma solução com PR é a construção deste modelo, sob o qual as restrições serão postadas, e avaliadas segundo um mecanismo *completo* de busca. Em PR o tema da modelagem é investigado em [18] e buscas em [21, 8].

1.1.2 Problemas de Satisfação de Restrições

Um problema combinatorial clássico é apresentado por um conjunto de variáveis de um sistema, as quais serão instanciadas por objetos de domínios, segundo um conjunto de relações, as quais representam o relacionamento entre os objetos. A tarefa combinatorial é dada pela ação de instanciar estes objetos as variáveis, de tal modo que todas as relações sejam satisfeitas.

A esta classe de problemas combinatoriais é conhecida como *Problemas de Satisfação de Restrições* (PSRs). A resolução dos PSR's constituem em encontrar valores as variáveis respeitando ou satisfazendo suas restrições. Para esta classe de problemas lança-se mão do uso da *Programação por Restrições* (PPR ou PR), ou seja, uma técnica que utiliza uma teoria e ferramentas próprias de programação. A PPR é uma forma de aplicar os conceitos de variáveis, domínios e restrições, via esta teoria específica de programação.

Um PSR é tipicamente um problema *NP-Completo* [15]. O desafio de todo o processamento por restrições está em gerar algoritmos que resolvam esta classe de problemas em um tempo computacional aceitável. Invariavelmente, alguns destes problemas NP, podem apresentar uma complexidade espacial considerável, assim passam para classe P-SPACE [17]. Dado este aspecto combinatorial e de complexidade NP, esta passa ter interesse por

outras àreas da pesquisa que lidam buscas *heurísticas*, tais como a Computação Evolutiva (CE), ou ainda buscas *completas* com a IA clássica e a Pesquisa Operacional (PO), etc. Esta situaçao é ilustrada pela figura 1.1.

Figura 1.1: Família dos problemas do tipo satisfação de restrições

A Programação por Restriçõespor sua vez, a exemplo da IA, CE, PO, etc, apresenta várias outras subdivisões e sub-áreas de interesse. Uma delas ela é a *Programação em Lógica com Restrições* (PLR) um dos segmentos a serem atacados nesta pesquisa. A PLR tem na lógica de primeira-ordem o seu modelo computacional, o qual é calculado a partir de buscas exaustivas sobre os seus *modelos consistentes* (modelos de Herbrand).

Uma das partes mais instigantes sobre os PSRs é que os mesmo são onipresentes em problemas do mundo real. Alguns destes problemas são discutidos em [15]. Destancam-se os problemas de escalonamento, planejamento, roteamento, contenção, alocação, etc.

As restrições podem ser consideradas como informações e dados há um programa por restrições. Estas visam limitar o *espaço de busca* e descrevem propriedades de variáveis/objetos e o relacionamento entre eles. As restrições são formalizadas como uma relação entre os objetos e esses são modelados como variáveis [6].

A relação existente entre os problemas de satisfação de restrições e a programação por restrições é expressa pela figura 1.1. Portanto, pode-se considerar que a PR está contida em PSR. Ou seja, a PR é um método que pode ser aplicado para encontrar a solução de problemas do tipo PSR.

Na PR, se a abordagem for feita via programação em lógica, então tem-se a PLR. A PLR é atrativa sob os seguintes requisitos metodológicos:

- Adequação a representação do conhecimento, caso este seja construído em lógica formal;
- Rápida prototipação e consequentemente baixo custo de desenvolvimento;
- Visão declarativa de suas restrições, possibilitando uma facilidade quanto aos testes e depuração;
- Flexibilidade na codificação dos algoritmos por abstrair características de programação em lógica.

Contudo, a Programação em Lógica com Restrições é resultado da utilização do paradigma da programação em lógica somado à programação por restrições. A PLR está contida no subconjunto de técnicas que fazem uso da PL para resolver problemas do tipo PSR. As vantagens de se utilizar a programação em lógica por restrições (PLR) estão em: modelar problemas de forma declarativa com uma sólida base matemática, propagação dos efeitos das decisões utilizando algoritmos eficientes e busca por soluções ótimas [6]. Desde o início

dos anos 90, a programação baseada em restrições tem tido sucesso comercial e industrial [16]. Em 1996, o mundo gerou, utilizando tecnologia de restrições um valor estimado de 100 milhões de dólares [6].

refazer as secoes anteriores

1.2 Complexidade

Refazer o Conteudo anterior e acrescentar o tema sobre complexidade e problemas P e NP e com isto o que a PR resolve

1.3 Árvores de Buscas: Ilustrando a Complexidade

acrescentar conteúdo dos slides do curso ...

1.4 Conclusões Parciais

Faltando .... e melhorar a apresentacao acima

2

Programação por Restrições

Este capítulo visa prover uma base de como a PR funciona.

2.1 Apresentando a PR

A Programação por Restrições (PR) é um paradigma de programação, tal como o paradigma imperativo, o orientado a objetos, o funcional, o lógico e outros. A idéia geral da PR, conforme [7], é de resolver problemas simplesmente declarando restrições que devem ser satisfeitas pelas soluções destes.

De acordo com [1], a PR já foi aplicada com sucesso em diversas áreas, como biologia molecular, engenharia elétrica, pesquisa operacional e análise numérica. Entre as principais aplicações, pode-se citar sistemas de suporte à decisão para problemas de escalonamento e alocação de recursos [7].

Problemas que hoje são identificados como sendo de satisfação de restrições, como por exemplo programar os horários de trabalho em uma empresa, sempre estiveram naturalmente presentes de alguma forma com a humanidade. Entretanto, métodos específicos para solucionar este tipo de problema começaram a surgir no meio acadêmico apenas nas décadas de 1950 e 1960, apesar de que técnicas como o *backtracking* (um método de busca exaustiva refinado) já eram utilizadas de forma recreativa desde o século XIX [14].

A Seção 2.2 apresenta alguns conceitos básicos deste paradigma, de forma a esclarecer como este funciona e o que o difere do paradigma imperativo. A Seção 2.4 esclarece as propriedades da modelagem de problemas por meio de uma linguagem de PR. A Seção 3.1 apresenta a linguagem MiniZinc e demonstra alguns recursos da linguagem com o auxílio de um exemplo.

2.2 Ilustrando os Conceitos

Algumas definições de termos comumente utilizados na área são apresentadas em [10]. Conforme este, uma restrição expressa uma relação desejada entre um ou mais objetos. Uma linguagem de PR é uma linguagem que possibilita descrever os objetos e as restrições.

Um programa feito em uma linguagem de PR define um conjunto de objetos e um conjunto de restrições sobre estes objetos. Um sistema de satisfação de restrições encontra soluções para programas feitos em uma linguagem de PR, ou seja, atribui valores aos objetos de forma que todas as restrições sejam satisfeitas.

Conforme [11], as restrições presentes no mundo real podem ser modeladas por meio de restrições em linguagem matemática. Desta forma, os objetos que tais restrições relacionam são também de cunho matemático, tais como variáveis e números.

Um exemplo simples apresentado em [10] é a conversão de temperaturas. Por exemplo, a restrição:

$$C = (F - 32) \times \frac{5}{9}$$

define o relacionamento entre temperaturas em Fahrenheit e Celsius, representadas respectivamente pelas variáveis C e F.

Em [10], também é reforçado o fato de que o sinal de igualdade em linguagens de PR possui o mesmo sentido da igualdade matemática. Isto difere-se de grande parte das linguagens imperativas (por exemplo C, Java e Python), nas quais o sinal de igualdade representa a operação de atribuir um valor a uma variável. Desta forma, em uma linguagem de PR, a restrição acima poderia ser reescrita, por exemplo, da seguinte forma:

$$9 \times C = 5 \times (F - 32)$$

Como tal restrição descreve o relacionamento entre as variáveis C e F, a partir do momento em que uma das variáveis receber um valor, o valor da outra variável já pode ser calculado. Além disto, caso fosse necessário realizar conversões para a escala Kelvin, bastaria adicionar a seguinte restrição:

$$K = C - 273$$

em que K é a variável que representa a temperatura em Kelvin. Assim como na restrição anterior, caso o valor de uma das variáveis for conhecido, o valor da outra pode ser calculado. Além disto, as restrições pertencentes ao conjunto de restrições de um dado programa são dependentes entre si. Desta forma, apenas com estas duas restrições é possível converter temperaturas entre Kelvin e Fahrenheit, sem explicitar a fórmula de conversão entre tais unidades [10].

Entretanto, as restrições não se limitam à equações lineares. Os tipos de restrição e de domínios de variáveis variam dependendo da linguagem. Os tipos comuns de domínio de variáveis são, por exemplo, inteiros, reais, booleanos, conjuntos, vetores, e outros. Também se faz possível declarar inequações, por exemplo, por meio dos operadores $\geq$ e $\neq$.

As operações possíveis entre as variáveis também variam, mas normalmente se faz possível utilizar as operações aritméticas básicas, como $+, -, /, \times$, para inteiros e reais, operadores lógicos, como $\wedge, \vee, \rightarrow, \leftrightarrow$, para booleanos, operações como $\cup$ e $\cap$ para conjuntos, entre outros.

Outro tipo de domínio de grande importância para a PR são os domínios finitos. Os valores possíveis que uma variável de domínio finito pode receber são restritos a um conjunto

finito de valores. Um exemplo clássico de domínio finito é o domínio booleano, no qual as variáveis só podem receber os valores *true* ou *false*. Outro exemplo são os intervalos de valores inteiros, como $[1, 10]$, por exemplo.

Os domínios finitos são amplamente utilizados na PR por permitirem ao programador modelar de forma natural problemas que envolvem escolhas, representando cada uma das possíveis escolhas por meio dos valores presentes no domínio [11].

2.3 Elementos ou Formalizando a PR

Esta seção é didaticamente apresentada por alguns fundamentos da PR, pois, estes elementos se descrevem as restrições e declarações sob um problema. Os problemas devem ser modelados, categorizados, ou seja, representados de forma a fazer com que se possa aplicar um determinado método de busca para encontrar a(s) solução(ões). Diversas são as técnicas encontradas na IA para a modelagem e resolução de problemas [16].

Uma delas é a aplicação de técnicas para a solução de Problemas de Satisfação de Restrições - PSR, ou, problemas que podem ser resolvidos pelo uso de restrições.

Um problema de satisfação de restrições (PSR) é uma tupla (V, D, R) onde [2]:

- V é um conjunto de n variáveis $\{x_1, ..., x_n\}$. Por convenção, variáveis são representadas pelas letras x, y, z, etc, indexadas por i se for caso de um número significativo das mesmas; fato que é regra geral.

- $D = \{D_1, ..., D_n\}$ é um conjunto de domínios. Onde cada componente D_i é o domínio que contém todos os possíveis valores que se podem atribuir à variável x_i.

- R é um conjunto finito de restrições. Cada restrição n-ária (R_n) está definida sobre um conjunto de variáveis $\{x_1, ..., x_x\}$ restringindo os valores que as variáveis podem, simultaneamente possuir. Este conjunto é dado por: $R = \{r_1, r_2, ..., r_m\}$

Em uma descrição, pode-se conceituar um PSR como um problema que pode ser composto por um conjunto de variáveis, cada qual associada à um domínio e um conjunto de restrições que se aplica aos valores que as variáveis podem, simultaneamente, serem atribuídas ou assumirem. A tarefa está em descobrir um valor no domínio para cada variável que satisfaça todas as restrições [20]. Um conjunto de variáveis $V = x_1, ..., x_n$, em associação com o domínio $D_1, ..., D_n$, respectivamente, possui uma *relação* R com um conjunto de variáveis que resulta em um subconjunto de um produto destes domínios. O conjunto de variáveis no qual a restrição está definida é chamado de *escopo* da relação ora da *restrição*, denotado por *grau* ou *escopo* de variável x_i no conjunto R. Cada relação que um subconjunto de mesmo produto $D_1 \quad \times ... \times \quad D_n$ de n domínios é dita: *aridade n*. Se $n = 1, 2$ ou 3, então a relação é chamada *unária, binária* e *ternária* respectivamente. Se $R = D_1 \quad \times ... \times \quad D_n$, então R é chamado de relação *universal*.

A PR relaciona restrições e declarações sobre um problema. Os problemas devem ser modelados, ou seja, representados de modo que se possa aplicar um determinado método

de busca para encontrar a(s) solução(ões). As diversas áreas da computação, matemática e PO, buscam apresentar técnicas para modelagem e resolução de problemas. Quanto a PR, esta metodologia é traduzida em construir um *modelo*, cuja a notação é dada pela tupla (V, D, R). Isto é: $Modelo = (V, D, R)$, onde:

V**:** um conjunto de variáveis do modelo do problema, $\{x_1, ..., x_n\}$;

D**:** um conjunto domínio(s), $\{D_1, ..., D_n\}$ em que as variáveis de V podem assumir valores;

R**:** tem-se no de m conjunto de restrições um mapeamento do tipo $(V \times D)^m \rightarrow V$. Assim, uma restrição é dada por: $r_j(x_1, ..., x_n)$

Logo, encontrar uma solução em PSR é logicamente expresso por:

$$\exists x_1 \exists x_2 ... \exists x_n (r_1(x_1, ..., x_n) \wedge r_2(x_1, ..., x_n) \wedge ... \wedge r_m(x_1, ..., x_n)) \tag{2.1}$$

Onde a sua interpretação lógica consistente é uma resposta ao problema. A descrição de sua solubilidade é análoga ao mundo de Herbrand [4, 13]. Cada restrição é aplicada a um subconjunto de variáveis, visando a satisfatibilidade em de seus valores. Uma atribuição é dita *consistente* se esta não violar nenhuma restrição. Assim, uma solução é encontrada quando todas as variáveis possuirem um valor consistente [16].

Logo, encontrar uma solução para um problema (p) em termos de (V, D, R), resume-se em encontrar uma construção de um *modelo* (M_p) para este problema. Logo, um modelo deve ser especificado por esta tupla, tal que $M_p = (V, D, R)$. Leia-se: *um modelo M para o problema p*. Em resumo, busca-se uma consistência da equação 2.1, computando-se sobre M_p de modo recursivo, aplicando suas restrições, propagação e expansão, sistematicamente sobre um procedimento de busca. Mairores detalhes: [2].

2.3.1 Definições

Para compreender a aplicação da PLR é necessário expor algumas definições próprias da área. Esta seção está organizada de forma a apresentar estas definições [5].

Conjunto: Um *conjunto* é uma coleção de objetos distintos e um objeto em uma coleção é chamado de um *membro* ou *elemento* de um conjunto.

Ordenação: Um conjunto não pode conter o mesmo objeto mais de uma vez, e estes elementos não são ordenados.

Variável: Uma variável possui uma coleção de valores, chamada domínio.

Domínio: Um domínio de uma variável é um conjunto que lista todas os objetos possíveis que a variável pode conter.

Tupla: Uma tupla é um a seqüência de objetos, não necessariamente distintos e um objeto em sequência é chamado de *componente*.

Restrições

As restrições são conduzem há um *encolhimento* no espaço de possibilidades (de estados) na busca por uma solução. A ordem pela qual as restrições são impostas não é relevante, mas sim, que ao final da conjunção dos termos seja atribuído o valor *verdadeiro*. As restrições possuem propriedades importantes a serem citadas [19]:

- Constitui uma *informação parcial*, haja vista que esta não pode, por si só, determinar o valor das variáveis do problema;
- As restrições são *aditivas*. Por exemplo: uma restrição $r_1 : X + Y \geq Z$ pode ser adicionada a uma outra restrição $r_2 : X + Y \leq W$.
- As restrições raramente são *independentes*. Geralmente compartilham variáveis, pois tratam sob um mesmo modelo. A combinação das restrições r_1 e r_2 resulta na obtenção de uma expressão algébrica do tipo: $Z \geq X + Y \leq W$.
- As restrições são ainda *não-direcionais*. Considerando a restrição $X + Y = Z$, esta pode ser utilizada para determinar a sua forma equivalente em $X \quad (X = Z - Y)$ ou em $Y \quad (Y = Z - X)$;
- As restrições são de natureza *declarativa* pelo fato de apenas denotarem as relações que devem ser asseguradas entre variáveis sem especificar um procedimento computacional para estabelecer esse relacionamento.

Estas características são típicas no uso de restrições em problemas do tipo PSR. A aplicação da restrição em um PR deve levar em consideração as variáveis do problema, bem como o domínio ao qual elas pertencem. A subseção 2.3.1, apresenta as definições de domínios.

Domínios

A maioria das linguagens orientadas a PSR possuem suporte a diversos tipos de domínios. Dentre elas destacam-se as restrições booleanas, domínios finitos, intervalos reais e termos lineares. Outros exemplos incluem listas, conjuntos finitos e árvores. Contudo, os principais domínios são visualizados na figura 2.3.1 [19].

Fazer uma figura nova

Os domínios correspondem ao tipo de valores que podem ser atribuídos às variáveis no momento da busca. Apesar da maioria dos problemas PSR poderem ser resolvidos utilizando domínios finitos, é importante relacionar aqui, este e outros domínios, tais como:

Domínios booleanos : são tratados por meta-interpretadores de restrições especializadas, podendo, no entanto, ser utilizados como um caso particular de restrições associadas à domínios finitos. As variáveis podem obter dois valores inteiros: 0 (falso) ou 1 (verdadeiro).

Domínios finitos : são utilizadas em muitas áreas do conhecimento. Para satisfação destas restrições usa-se uma combinação de técnicas para a preservação de consistência, propagação de valores e pesquisa com retrocesso [19, 2]. Cada variável possui associada a ela um conjunto finito de valores inteiros. Os valores do domínio inconsistentes são removidos do domínio das variáveis durante a propagação;

Números reais :, também conhecidas como intervalos reais, são equivalentes aos domínios finitos mas aqui são utilizados valores reais. Inclusive as técnicas de remoção de inconsistências são similares às técnicas usadas para com os domínios finitos. Outras técnicas matemáticas de diferenciação automática ou as séries de *Taylor* podem ser utilizadas [19];

Domínios lineares : ou restrições lineares, compreendem domínios construídos por meio de variáveis cujos valores são dados pelo conjunto dos números reais. Para este tipo de restrições têm sido implementados meta-interpretadores de restrições bastante eficientes que utilizam o algoritmo *simplex* como ponto de partida [19].

O uso do domínio PSR está diretamente ligado à modelagem desenvolvida pelo analista ou programador. Por outro lado, provavelmente, hoje em dia, mais de 95% de todas as aplicações que utilizam restrições são de domínios finitos [20, 3].

2.4 Modelagem

[9] afirma que os algoritmos possuem dois componentes distintos, a lógica e o controle. A lógica está relacionada com o que o algoritmo faz, o conhecimento que se possui sobre o problema. O controle, por sua vez, representa como tal lógica será utilizada para resolver o problema.

No paradigma imperativo, geralmente não há uma distinção clara entre tais componentes, visto que os problemas são resolvidos seguindo uma sequência de instruções, que fazem tanto parte do controle quanto da lógica propriamente dita [7].

Uma das vantagens do paradigma declarativo, do qual a PR faz parte, é a possibilidade de focar de forma praticamente exclusiva no componente lógico, sem se preocupar com o controle [7].

Desta forma, para se resolver um problema utilizando o paradigma de PR, basicamente se faz necessário apenas modelá-lo matematicamente em termos de váriaveis e restrições. Apesar de não ser uma tarefa trivial, em muitos casos fazer tal modelagem é muito mais simples do que desenvolver um algoritmo utilizando o paradigma imperativo, visto que neste último se faz necessário explicitar cada passo necessário para a resolução do problema, enquanto que com a PR basta descrever o problema.

Conforme [11], um problema de satisfação de restrições pode ser formalmente modelado por meio de uma restrição C sobre váriaveis $x_1, ..., x_n$, e um domínio D que mapeia cada variável x_i ao conjunto finito de valores que esta pode assumir. Tal restrição C é a conjunção de todas as restrições definidas.

Após o desenvolvimento de um programa em uma linguagem de PR, para se obter as soluções do problema, se faz necessário utilizar um sistema de satisfação de restrições (*solver*). De acordo com [14], tais sistemas utilizam diversos métodos para atribuir valores às variáveis de forma a satisfazer todas as restrições. Entre estes métodos, pode-se citar o *backtracking*, *branch and bound* e a propagação de restrições, que não serão apresentados em detalhes neste trabalho.

Em alguns casos, além de ser necessário satisfazer todas as restrições de um problema, também se faz preciso otimizar a solução deste. Nestes casos, além de se definir o conjunto de variáveis e de restrições, define-se também uma função objetivo a ser otimizada (maximizada ou minimizada). Tal função objetivo mapeia cada solução do problema em um valor real, possibilitando assim definir qual é a solução mais otimizada [1].

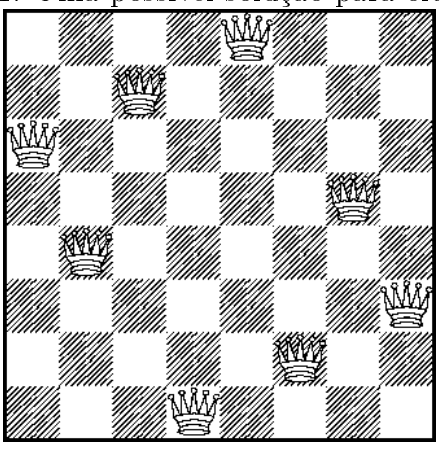
2.4.1 Exemplo de Modelagem

Um dos exemplos clássicos utilizados para ilustrar a PR é o famoso problema das n-rainhas. O objetivo deste problema é posicionar n rainhas em um tabuleiro $n \times n$ de xadrez, com $n > 3$, de tal forma que as rainhas não se ataquem mutuamente [1].

Uma das modelagens mais comuns para este problema, apresentado em [1], é representar a posição de cada uma das n rainhas por meio de variáveis $x_1, x_2, ..., x_n$, sendo o domínio de tais variáveis $[1, n]$. O valor de cada variável x_i indica qual é a linha em que está a rainha que se encontra na coluna i.

Para o tabuleiro apresentado na Figura 2.1, que demonstra uma possível solução para o problema com oito rainhas, o valor das variáveis x_i é respectivamente $6, 4, 7, 1, 8, 2, 5, 3$. Tais valores foram calculados assumindo a primeira coluna como sendo a mais a esquerda e a primeira linha como sendo a mais abaixo.

Figura 2.1: Uma possível solução para oito rainhas



Fonte: [1]

Para garantir que as rainhas sejam posicionadas de forma a não se atacarem, se faz necessário declarar um conjunto de restrições sobre as variáveis x_i. As desigualdades a

seguir são suficientes para garantir que os valores de tais variáveis formem uma configuração válida conforme o objetivo do problema, para $i \in [1, n-1]$ e $j \in [i+1, n]$:

- $x_i \neq x_j$
- $|x_i - x_j| \neq |i - j|$

A primeira desigualdade impede que duas ou mais rainhas sejam posicionadas na mesma linha. Tal desigualdade gera um total de $\frac{n(n-1)}{2}$ restrições ao se aplicar todos os possíveis valores de i e j. Todas estas restrições juntas garantem que todas as variáveis assumirão valores distintos entre si.

A segunda desigualdade impede que duas ou mais rainhas sejam posicionadas na mesma diagonal. Assim como para a primeira desigualdade, esta também gera um total de $\frac{n(n-1)}{2}$ restrições ao se aplicar todos os possíveis valores de i e j, gerando assim um conjunto com $n(n-1)$ restrições ao todo.

Não se faz necessário restrições para verificar se duas ou mais rainhas estão posicionadas na mesma coluna, visto que a forma como o tabuleiro foi modelado já impede naturalmente que tal situação aconteça.

Faltando .... e melhorar a apresentacao acima

2.5 Buscas

Faltando .... e melhorar a apresentacao acima

Referência para esta seção: [5] UM RESUMO + Taxonomia

2.6 Acelerando as Buscas

Faltando .... e melhorar a apresentacao acima

2.7 Conclusões Parciais

Faltando .... e melhorar a apresentacao acima

3

A Linguagem MiniZinc

3.1 Apresentando o MiniZinc

Conforme [12], antes do advento da linguagem de programação MiniZinc, não existia uma linguagem padrão para a modelagem de problemas de programação por restrições. Praticamente cada *solver* possuia sua própria linguagem de modelagem. Isto dificultava a tarefa de realizar experimentos para comparar o desempenho de diferentes *solvers* para um determinado problema.

O MiniZinc surgiu com a proposta de criar uma linguagem padrão de modelagem, na qual um mesmo modelo pode ser utilizado por diversos *solvers*.

Faltando descrever o que sao estes solvers....

Além disto, outras de suas qualidades são a simplicidade, a expressividade e a facilidade de implementação [12]. A simplicidade se encontra principalmente na sintaxe, e a expressividade é dada por permitir modelar diversos tipos de problemas por meio dos recursos disponibilizados.

3.2 Elementos da Linguagem

3.2.1 Um Paradigma Computacional

$$Modelo + Dados = A_1 \wedge A_2 \wedge A_3 \wedge .... \wedge A_n$$

- A_i : são assertivas declaradas (declarações de restrições) sobre o problema
- Logo, esta é uma linguagem que visa construir rapidamente modelos sobre os problemas reais.
- Detalhe é que estes **modelos** são computáveis!

3.2.2 Estrutura de um Programa em MiniZinc

Faltando uma figura

3.2.3 Constantes ou Parâmetros

As constantes são similares às váriaveis de linguagens de programação comuns. Entretanto, só é permitido atribuir um valor a um parâmetro uma vez.

Faltando ....

3.2.4 Variáveis

Variáveis e Variáveis de Decisão (discutidas ao longo do texto) são mais próximas ao conceito de incógnitas da Matemática. O valor de uma váriavel de decisão é escolhido pelo Minizinc para atender todas as restrições estabelecidas.

Faltando ....

Booleanas: Faltando ....define e exemplo

Inteiras:

Reais:

Vetores ou Arrays:

Conjuntos:

Enumeradores:

Contudo, caso estas variáveis exibam um domínio específico, ou restrito, que é o caso de interesse, estas variáveis são conhecidas como *variáveis de restrição*. Estas assumem valores que são *descobertos* dentro de um domínio de valores sob um conjunto de restrições que é o *modelo computado*.

3.2.5 Restrições ou Declarações Lógicas

Faltando ....

3.2.6 Satisfatibilidade e Otimização

Faltando ....

3.2.7 Saídas

Faltando ....

3.2.8 Instalação e Uso

Revisao ....

1. Baixar o arquivo .tar.gz
   (http://www.minizinc.org/g12distrib.html)
2. Extrair a pasta.
3. Executar o arquivo SETUP presente na pasta.
4. Alterar a variável PATH, execute o comando:
   export PATH=$PATH:/<diretório da pasta do minizinc>/bin

3.3 Um Exemplo

Esta seção apresenta um exemplo completo, o qual resume os conceitos apresentados anteriormente. Este exemplo provê noções básicas para compreender e desenvolver programas na linguagem MiniZinc, sendo tais programas comumente chamados de modelos. Para isto, é tomado como exemplo um modelo feito em MiniZinc, apresentado na Figura 3.1.

```
1  include "globals.mzn";
2
3  int: n = 8;
4  array[1..n] of var 1..n: pos_rainhas;
5
6  constraint n > 3;
7
8  constraint alldifferent(pos_rainhas);
9
10 constraint forall(i in 1..n-1) (
11               forall(j in i+1..n) (
12                 abs(pos_rainhas[i] - pos_rainhas[j]) != abs(i -
       j)
13               )
14             );
15
16 solve satisfy;
17
18 output [show(pos_rainhas)];
```

Listing 3.1: Exemplo de um modelo em MiniZinc

Este é um modelo para o problema das n-rainhas, apresentado na Subseção 2.4.1. A primeira linha contém o comando *include*, que funciona de forma análoga às outras linguagens de programação, como C. Este comando serve para possibilitar a divisão de um mesmo

modelo em diversos arquivos e utilizar bibliotecas externas. Neste caso, está sendo incluída a biblioteca de restrições globais por meio do arquivo *globals.mzn*, com o intuito de utilizar a restrição *alldifferent*, pertencente a tal biblioteca, no modelo.

Na linha 3 está sendo declarada a variável n e atribuindo a esta o valor 8. Tal variável representa o tamanho do tabuleiro, e por consequência a quantidade de rainhas presentes neste. Na linguagem MiniZinc as variáveis podem ser parâmetros ou variáveis de decisão. Os parâmetros são variáveis de valor fixo e conhecido. Já as variáveis de decisão são variáveis cujo valor será atribuído pelo *solver*, de forma a satisfazer todas as restrições.

Os valores dos parâmetros podem ser definidos tanto no próprio modelo, como no caso do modelo apresentado na Figura 3.2, ou em um arquivo externo de extensão *.dzn*. Caso o usuário esteja utilizando a IDE do MiniZinc, também é possível fornecer os valores dos parâmetros por meio de uma interface gráfica. Tal interface é exibida ao usuário quando este pressiona o botão para executar o modelo e os valores dos parâmetros não são fornecidos no código.

Ao se declarar uma variável em MiniZinc, se faz preciso informar o tipo desta e se esta é um parâmetro ou uma variável de decisão. Entre os tipos de variável disponíveis estão, por exemplo, *int*, *float* e *bool*.

Para se especificar que uma váriavel é de decisão, é necessário utilizar o prefixo *var* antes do tipo da variável. Caso não haja um prefixo antes do tipo de uma variável, o MiniZinc irá considerar esta como sendo um parâmetro, apesar de ser possível utilizar o prefixo *par* para se explicitar que esta é um parâmetro. Desta forma, a variável n declarada na linha 3 é um parâmetro, visto que não há o prefixo *var* antes desta.

Na linha 4 há a declaração do vetor *pos_ rainhas*, que neste modelo representa as posições das rainhas. Neste vetor, a i-ésima posição representa a linha na qual se encontra a rainha que está na i-ésima coluna do tabuleiro.

Para se declarar um vetor em MiniZinc, se faz necessário definir o intervalo de índices de cada uma de suas dimensões, o tipo dos seus elementos e se estes são parâmetros ou variáveis de decisão.

No caso do vetor *pos_ rainhas*, há apenas uma dimensão, sendo que os índices variam de 1 até o valor do parâmetro n. Os índices possíveis de cada uma das dimensões do vetor são especificados dentro dos colchetes, após a palavra chave *array*.

Os elementos do vetor *pos_ rainhas* são variáveis de decisão, visto que o intuito do modelo é justamente encontrar as posições em que as rainhas devem ser posicionadas. Neste caso, os valores que os elementos podem assumir estão restringidos a um domínio finito, que é o intervalo inteiro que varia de 1 até n.

A linha 6 apresenta a primeira restrição do modelo, que garante que o valor do parâmetro n informado pelo usuário é maior que três. As restrições em MiniZinc devem iniciar com a palavra chave *constraint* e devem ser expressões booleanas, isto é, devem resultar em verdadeiro ou falso. Tais expressões podem envolver parâmetros, variáveis de decisão e constantes, que devem ser relacionados por meio de algum operador de relação, como $>$, $<=$, $=$ e $!=$.

Para facilitar a modularização dos modelos, o MiniZinc possibilita a definição de predicados, e a partir da versão 2.0, também permite a definição de funções. A linha 8 mostra

o predicado *alldifferent*, que faz parte da biblioteca de restrições globais do MiniZinc. Este predicado recebe um vetor de inteiros e garante que os valores dos elementos deste vetor são distintos entre si. Esta restrição é equivalente à primeira desigualdade apresentada na Subseção 2.4.1.

Na linha 10 é apresentada uma restrição um pouco mais complexa. Esta faz uso do *forall*, que funciona de forma análoga ao quantificador universal ∀. Em MiniZinc, utiliza-se o *forall* principalmente quando deseja-se agrupar diversas restrições semelhantes, de modo a simplificar a leitura e o desenvolvimento do modelo. Normalmente tais restrições semelhantes estão vinculadas a vetores ou conjuntos, e se faz possível generalizar estas restrições utilizando iteradores.

Uma das formas de se utilizar o *forall* é especificando um iterador, o conjunto de possíveis valores que este pode assumir e a restrição generalizada em termos deste iterador. Desta forma, para cada um dos valores que o iterador pode assumir será gerada uma expressão *booleana*, em que o iterador será substituido pelo valor assumido. Todas estas expressões geradas são unidas em uma única restrição, sendo que tal união é feita por meio da conjunção de todas as expressões. Desta forma, para que a restrição criada seja satisfeita, todas as expressões *booleanas* geradas devem resultar em verdadeiro.

A linguagem MiniZinc também oferece o *exists*, que funciona de forma praticamente igual ao *forall*, sendo que a única diferença entre estes está na forma como as expressões *booleanas* são unidas, que neste caso é por meio de disjunções. Assim, para que a restrição criada pelo *exists* seja satisfeita, basta que uma das expressões que compõem a restrição seja satisfeita.

No caso da restrição presente na linha 10, o *forall* é utilizado duas vezes, de forma aninhada. Isto é feito pois se faz necessário garantir que todos os pares possíveis de rainhas não estão se atacando mutuamente.

A linha 12 apresenta a expressão *booleana* que verifica se um par de rainhas está ou não se atacando, de forma equivalente à segunda desigualdade apresentada na Seção 2.4.1. A função *abs* retorna o valor absoluto, ou o módulo, de um inteiro.

Na linha 16, há a indicação de que o problema é de satisfação de restrições. Caso fosse um problema de otimização, seria necessário substituir a palavra chave *satisfy* por *maximize* ou *minimize*, seguido da função que se deseja maximizar ou minimizar.

Por fim, na linha 18 há o comando *output*, pelo qual pode-se imprimir na tela os resultados obtidos. Tal comando é bastante flexível, de forma que saídas complexas podem também ser geradas. Para este problema seria possível, por exemplo, imprimir o tabuleiro utilizando algum caracter para representar as rainhas.

Na biblioteca apresentada no capítulo 4, as funcionalidades são implementadas por meio de funções. Isto é feito para que seja possível utilizar esta biblioteca em qualquer modelo, bastando para isto incluir um arquivo, como foi feito neste exemplo para utilizar a biblioteca de restrições globais. Para exemplificar a definição de funções, a Figura 3.2 apresenta um modelo equivalente ao apresentado anteriormente, entretanto, utilizando uma função para a resolução do problema.

```
1 include "globals.mzn";
2
```

```
3  int: n = 8;
4  array[1..n] of var 1..n: pos_rainhas;
5
6  function array[int] of var int: n_queens(int: num_of_queens) =
7  let {
8    array[1..num_of_queens] of var 1..num_of_queens: queens;
9    constraint num_of_queens > 3;
10   constraint alldifferent(queens);
11   constraint forall(i in 1..(num_of_queens-1)) (
12                 forall(j in (i+1)..num_of_queens) (
13                   abs(queens[i] - queens[j]) != abs(i - j)
14                 )
15               );
16 } in queens;
17
18 constraint pos_rainhas = n_queens(n);
19
20 solve satisfy;
21
22 output [show(pos_rainhas)];
```

Listing 3.2: Exemplo de utilização de função

Na linha 6 deste exemplo, há a declaração da função. Esta é iniciada com a palavra chave *function*, que é sucedida pela declaração do tipo de retorno da função. Em MiniZinc, toda função precisa obrigatoriamente de um retorno, visto que quando este não se faz necessário pode-se substituir o uso de uma função pelo uso de um predicado.

Após o tipo de retorno, há o identificador da função, que é utilizado na hora de realizar chamadas à função. Após o identificador, há a lista de argumentos da função. Para cada argumento é especificado um tipo e um identificador.

Neste exemplo, a função é chamada de *n_queens* e recebe como argumento o tamanho do tabuleiro desejado, identificado como *num_of_queens*. O retorno desta função é um vetor no qual os elementos são variáveis de decisão inteiras. Este vetor representa uma possível solução para o problema, de acordo com a modelagem proposta anteriormente.

A estrutura *let{} in* presente na função, permite o uso de variáveis locais. De forma geral, quando esta estrutura é utilizada em funções, pode-se comparar o conteúdo presente dentro do *let* como sendo o corpo da função, e a expressão após o *in* como sendo o retorno desta.

Após a conclusão de um modelo, se faz preciso utilizar um *solver* para encontrar os valores das variáveis de decisão, de forma a satisfazer todas as restrições. Isto pode ser feito, por exemplo, pela própria IDE fornecida pelo MiniZinc, na qual pode-se editar os modelos e avaliá-los. Caso não haja uma solução possível, o MiniZinc apresenta uma mensagem informando que o modelo é insatisfatível, ou seja, não existe uma atribuição de valores às variáveis de decisão que satisfaça as restrições estabelecidas. Há ainda casos em que o

modelo em questão pode ter muitas possibilidades de soluções para serem avaliadas, gerando uma explosão combinatorial, o que pode fazer com que o *solver* não consiga encontrar uma solução em um tempo aceitável.

Para a implementação da biblioteca, alguns outros recursos do MiniZinc que não são explicados nesta seção são utilizados. Porém, tais recursos são explicados à medida em que são utilizados nas implementações.

3.4 Conclusões Parciais

Faltando .... e melhorar a apresentacao acima

4

Modelagem de Problemas

Neste capítulo são apresentados alguns estudos de casos.

4.1 Os Modelos em Estudo

XXXXXXXXXXXXXXXXXXXXX ... bons modelos e classicos interessantes ....

4.2 Cabo de Guerra

4.2.1 Descrição

Melhorar o enunciado ... mas o outros estão melhores

Várias crianças se encontram para brincadeira do cabo-de-guerra no pátio da escola. Como será feita a divisão entre as duas equipes? "*Por peso*" grita o mais eufórico. Que seja feita a divisão dos times sobre uma sequência/lista de peso tal como:

$Joao_1$	$Pedro_2$	$Manoel_3$	....	$Zeca_n$
45	39	79	....	42

➠ Preencha a tabela acima com inteiros e valores de sua família.

➠ Sim, por peso, todos concordaram, "*exceto que a divisão deveria respeitar o critério* $|N_A - N_B| \leq 1$", disse o mais cauteloso. Sim, nenhum time poderia duas crianças a mais que ou outro time.

4.2.2 Especificação

Que seja feita a divisão:

$Joao_1$	$Pedro_2$	$Manoel_3$	....	$Zeca_n$
45	39	79	....	42

- Divisão por peso
- Respeitar critérios como: $|N_A - N_B| \leq 1$
- Todos devem brincar
- Bem, esta simples **restrição** ($|N_A - N_B| \leq 1$), de nosso cotidiano tornou um simples problema em mais uma questão combinatória. Um arranjo da ordem de $\frac{n!}{(n/2)!}$. Casualmente, nada trivial para grandes valores!

➠ Dois detalhes:

1. Na tabela de pesos, use valores **inteiros**;
2. Use os pesos de seus familiares para completar esta tabela com um quantidade significativa;
3. No lugar de *array* como estrutura base, use *sets* para armazenar e manipular estes valores. Com certeza ficará mais *elegante*, e possivelmente mais ineficiente. Teste e comprove!

4.2.3 Modelagem

- Usando uma variável de decisão: análogo a árvore do SAT

Nomes (n_i):	n_1	n_2	n_3	....	n_n
Peso (p_i):	45	39	79	....	42
Binária (x_i):	0/1	0/1	0/1	....	0/1

- Assim $N_A \approx N/2$, $N_B \approx N/2$ e $|N_A - N_B| \leq 1$
- $x_i = 0$: n_i fica para o time A
- $x_i = 1$: n_i fica para o time B
- Logo a soma:

$$\sum_{i=1}^{n} x_i p_i$$

é o peso total do time B (P_B)

- Falta encontrar peso total do time A (P_A), dado por:
- $P_A = P_{total} - P_B$
- ou

$$P_A = \sum_{i=1}^{n} p_i - \sum_{i=1}^{n} x_i p_i$$

- Finalmente, aplicar uma minimização na diferença: $|P_A - P_B|$

4.2.4 Estratégia

Uma árvore de decisão binária ..... descreva como voce implementou ou a fundamentacao

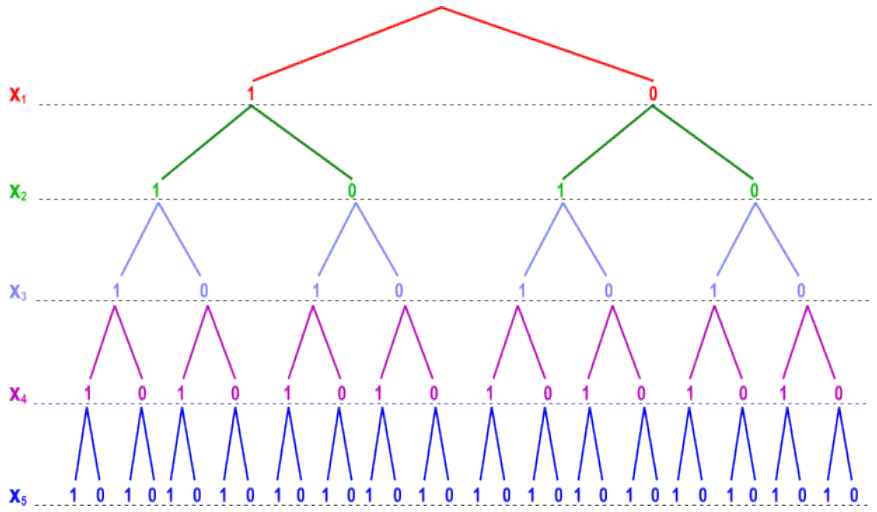


Figura 4.1: Se $x_i = 0$, então n_i segue para o time A, caso $x_i = 1$, então n_i vai para o time B

4.2.5 Implementação

O código fonte se encontra em:
`https://github.com/claudiosa/CCS/tree/master/minizinc/cabo_de_guerra.mzn`

4.2.6 Resultados e Análise

Considerando pesos aleatórios de 1 a 150 para as pessoas
Usando um *solver* médio do Minizinc (*G12 lazyfd*) padrão:
Referência: cpu 4-core, 4 G ram, SO: Linux-Debian

Tabela 4.1: Resultados .....

n	tempo	P_A	P_B
5	40msec	276	278
10	46msec	518	519
25	98msec	1198	1197
50	411msec	2290	2291
75	**2s 485msec**	3133	3133
100	470msec	4142	4142
125	**7s 2msec**	4992	4992
150	605msec	5823	5823
175	642msec	6777	6778
200	$>$ 10min	–	–

Bibliografia

[1] K. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[2] Krzysztof Apt. *Principles of Constraint Programming.* Cambridge University Press, New York, NY, USA, 2003.

[3] Krzysztof Apt. *Principles of Constraint Programming.* Cambridge University Press, New York, NY, USA, 2003.

[4] C.K. Chang and R.C. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press - New York, 1973.

[5] Rina Dechter. *Constraint Processing.* Morgan Kaufmann, 2003.

[6] Thom Fruewirth and Slim Abdennadher. *Essentials of Constraint Programming.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[7] T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming.* Cognitive Technologies. Springer, 2003.

[8] Holger H. Hoos and Edward Tsang. *Local Search Methods*, chapter 5, pages 135 – 167. Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA, 2006. Edited by Francesca Rossi, Peter van Beek and Toby Walsh.

[9] Robert Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, July 1979.

[10] W. Leler. *Constraint programming languages: their specification and generation.* Addison-Wesley series in computer science and information processing. Addison-Wesley Longman, Incorporated, 1988.

[11] K. Marriott and P.J. Stuckey. *Programming with Constraints: An Introduction.* Adaptive Computation and Machine. MIT Press, 1998.

[12] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *In: Proc. of 13th International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[13] Ulf Nilsson and Jan Maluszyn. *Logic, Programming and Prolog.* John Wiley & Sons, 2nd. edition, Disponível para download: http://www.ida.liu.se/ ulfni/lpp/, 2000.

[14] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming.* Foundations of Artificial Intelligence. Elsevier Science, 2006.

[15] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence).* Elsevier Science Inc., New York, NY, USA, 2006. http://portal.acm.org/citation.cfm?id=1207782#.

[16] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.).* Pearson Education, 2010.

[17] Michael Sipser. *Introduction to the Theory of Computation (3rd Edition).* Cengage Learning, Boston, MA, USA, 2012.

[18] Barbara M. Smith. *Modelling*, chapter 11, pages 377–406. Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA, 2006. Edited by Francesca Rossi, Peter van Beek and Toby Walsh.

[19] Igor Ribeiro Sucupira. Programação por propagação de restrições: Teoria e aplicações. Master's thesis, Dissertação. Universidade do Estado de São Paulo. IME: USP, 2003. ver detalhes nesta referência.

[20] Edward Tsang. *Foundations Of Constraint Satisfaction.* Academic Press Limited, 1993. Department of Computer Science – University of Essex – Colchester – Essex – UK.

[21] Peter van Beek. *Backtracking Search Algorithms*, chapter 4, pages 85–134. Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA, 2006. Edited by Francesca Rossi, Peter van Beek and Toby Walsh.