Modelos Computacionais com Programação por Restrições

Claudio Cesar de Sá e outros

18 de Janeiro 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
4.

1

Fundamentos de Complexidade de Problemas

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

Conteudo clássico sobre complexidade e problemas P e NP

2

Programação por Restrições

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 ideia geral da PR, conforme [2], 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 [2].

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 [7].

A Seção 2.1 apresenta alguns conceitos básicos deste paradigma, de forma a esclarecer como este funciona e o que o difere do paradigma imperativo. A Seção 2.2 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.

2.1 Conceitos Básicos

Algumas definições de termos comumente utilizados na área são apresentadas em [4]. 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 [5], 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 [4] 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 [4], 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 [4].

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 [5].

2.2 Modelagem

[3] 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 [2].

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 [2].

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 [5], 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 [7], 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].

2.2.1 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.

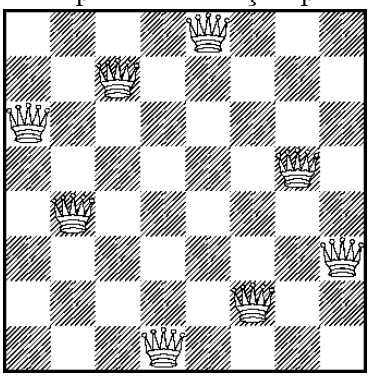
Para o tabuleiro apresentado na Figura 2.1, que demonstra uma possível solução para o problema com oito rainhas, o valor das variáveis x_i é respectivamente $6, 4, 7, 1, 8, 2, 5, 3$. Tais valores foram calculados assumindo a primeira coluna como sendo a mais a esquerda e a primeira linha como sendo a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

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

Figura 2.1: Uma possível solução para oito rainhas



Fonte: [1]

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.

3

A Linguagem de Programação MiniZinc

3.1 Apresentando o MiniZinc

Conforme [6], 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*. Além disto, outras de suas qualidades são a simplicidade, a expressividade e a facilidade de implementação [6]. A simplicidade se encontra principalmente na sintaxe, e a expressividade é dada por permitir modelar diversos tipos de problemas por meio dos recursos disponibilizados.

Esta seção apresenta de forma resumida algumas 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.

Este é um modelo para o problema das n-rainhas, apresentado na Subseção 2.2.1. 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

Figura 3.1: Exemplo de modelo em MiniZinc

```
include "globals.mzn";

int: n = 8;
array[1..n] of var 1..n: pos_rainhas;

constraint n > 3;

constraint alldifferent(pos_rainhas);

constraint forall(i in 1..n-1) (
              forall(j in i+1..n) (
                abs(pos_rainhas[i] - pos_rainhas[j])
   != abs(i - j)
              )
            );

solve satisfy;

output [show(pos_rainhas)];
```

codigos/nqueens.mzn

Fonte: Autoria própria

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 2.2.1.

Na linha 10 é apresentada uma restrição um pouco mais complexa. Esta faz uso do *forall*, que funciona de forma análoga ao quantificador universal $\forall$. 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 2.2.1. 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.

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

Figura 3.2: Exemplo de utilização de função

```
include "globals.mzn";

int: n = 8;
array[1..n] of var 1..n: pos_rainhas;

function array[int] of var int: n_queens(int:
   num_of_queens) =
let {
  array[1..num_of_queens] of var 1..num_of_queens:
   queens;
  constraint num_of_queens > 3;
  constraint alldifferent(queens);
  constraint forall(i in 1..(num_of_queens-1)) (
               forall(j in (i+1)..num_of_queens) (
                 abs(queens[i] - queens[j]) != abs(i
    - j)
               )
             );
} in queens;

constraint pos_rainhas = n_queens(n);

solve satisfy;

output [show(pos_rainhas)];
```

codigos/nqueens_fn.mzn

Fonte: Autoria própria

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 implementãções.

4

Modelagem de Problemas

Neste capítulo são apresentados alguns estudos de casos.

4.1 Cabo de Guerra

4.1.1 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.

4.1.2 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!

4.1.3 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: $|P_A - P_B|$

4.1.4 Estratégia

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

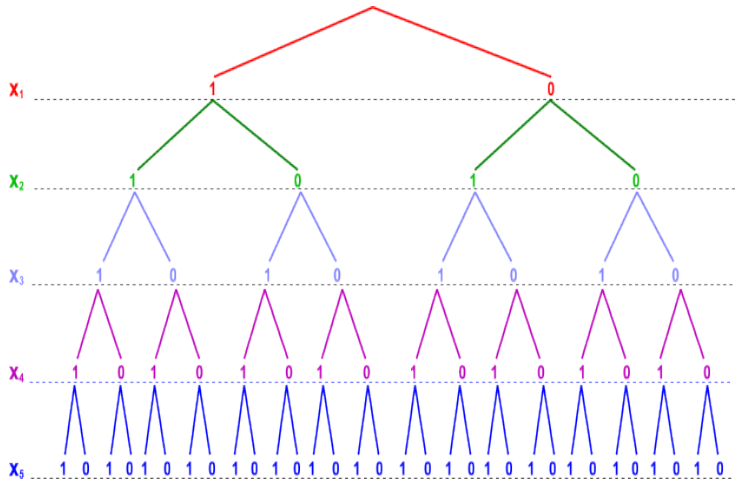


Figura 4.1: Se $x_i = 0$, então n_i segue para o time A, caso $x_i = 1$, então n_i vai para o time B

4.1.5 Implementação

O código fonte se encontra em:
`https://github.com/claudiosa/minizinc/blob/master/CEAVI_2015/cabo_de_guerra.mzn`

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

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

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

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

[6] 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.

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