Mini-Curso de Minizinc

Claudio Cesar de Sá
claudio.sa@udesc.br

Departamento de Ciência da Computação
Centro de Ciências e Tecnológias
Universidade do Estado de Santa Catarina

12 de agosto de 2016

Sumário

Notas

- Todos os códigos se encontram em: `https://github.com/claudiosa/CCS/minizinc`
- Agradecimentos

Problemas × Otimização

Trocar esta figura ....

Problemas e Otimização

Complexidade $\Leftrightarrow$ Encontrar soluções:

- Problemas complexos de interesse prático (e teórico): NPs $\Uparrow$
- Tentativas de soluções: diversas direções (teoria) e muitos paradigmas computacionais (práticas)
- Seguem desde um modelo matemático existente a um modelo empírico a ser descoberto. Exemplificando:

Problemas e Otimização

Complexidade $\Leftrightarrow$ Encontrar soluções:

- Problemas complexos de interesse prático (e teórico): NPs $\Uparrow$
- Tentativas de soluções: diversas direções (teoria) e muitos paradigmas computacionais (práticas)
- Seguem desde um modelo matemático existente a um modelo empírico a ser descoberto. Exemplificando:
  - Uma equação de regressão linear: $y = ax^2 + b$
  - ... até ...
  - Programação genética (evolução de um modelo)
- Problemas apresentam características comuns como: variáveis, domínios, restrições, espaços de estados (finitos e infinitos, contínuos e discretos) ...

Otimização

Complexidade $\Leftrightarrow$ Otimização:

- A área de **Otimização** tem uma divisão: Discreta ou Combinatória e Contínua ou Numérica (funções deriváveis)

Otimização

Complexidade ⇔ Otimização:

- A área de **Otimização** tem uma divisão: Discreta ou Combinatória e Contínua ou Numérica (funções deriváveis)

  Combinatória: Problemas definidos em um espaço de estados finitos (ou infinito mas enumerável)

  Numérica: Definidos em subespaços infinitos e não enumeráveis, como os números reais e complexos

- Difícil: problemas que tenham uma ordem maior ou igual a $2^{O(n)}$ são exponenciais, consequentemente, difíceis!

Como atacar estes problemas?

Técnicas:

Combinatória:

- Busca Local
- Métodos Gulosos: busca tipo subida a encosta (*hill-climbing*), recozimento simulado (*simulated annealing*), busca tabu, etc.
- Programação Dinâmica
- **Programação por Restrições** (PR)
- Redes de Fluxo
- .....

Numérica:

- Descida do Gradiente
- Gauss-Newton
- Lavemberg-Marquardt
- .....

Programação por Restrições (PR)

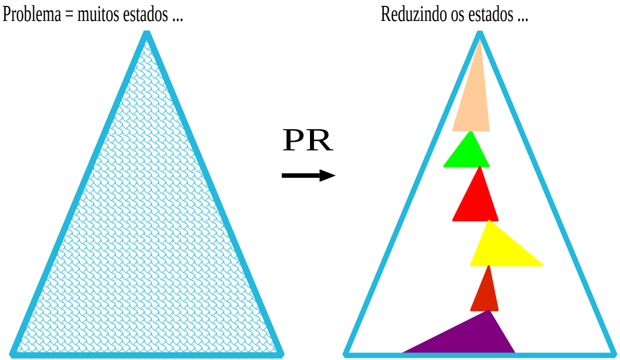


Figura: O *mar de estados* e a filtragem da PR

Onde o objetivo é:

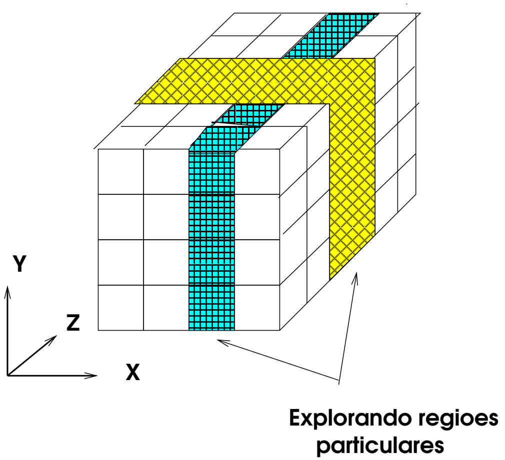


Figura: Operando com regiões específicas ou reduzidas

Redução em sub-problemas:

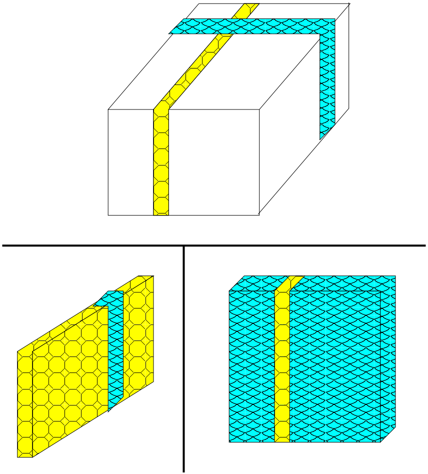
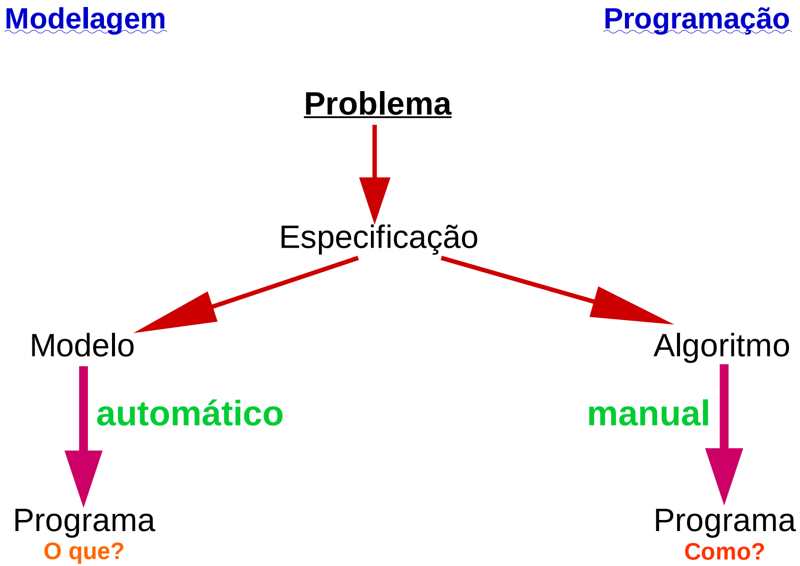


Figura: Redução de P em outros sub-problemas equivalentes

Construção de modelos e implementações:



Ferramentas: linguagens, tradutores e *solvers*:

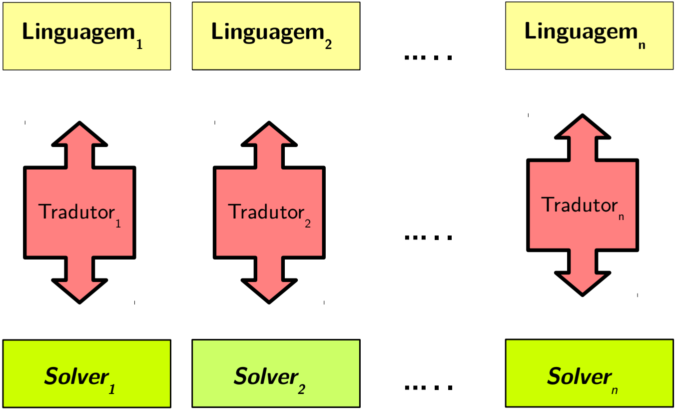


Figura: Linguagens, bibliotecas e *solvers* de propósitos diversos

Minizinc, tradutores e os *solvers*:

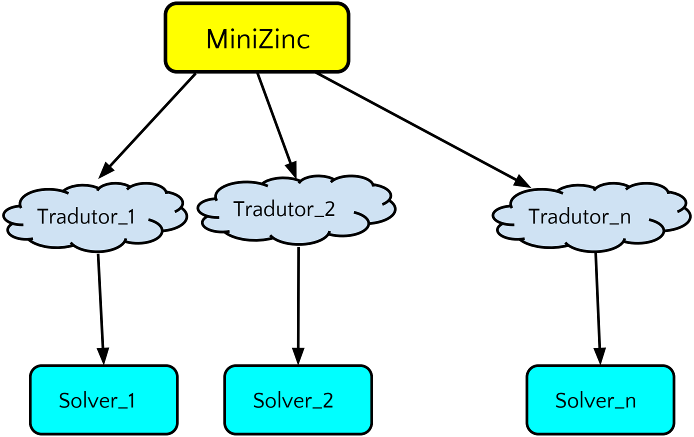


Figura: Há muitas conversores do FlatZinc para vários *solvers*

MiniZinc: uma proposta unificada

Histórico

➸ Em 2006 a comunidade de CP *Constraint Programming* discutiu a necessidade de uma linguagem unificada para seus modelos e pesquisas

Histórico

➳ Em 2006 a comunidade de CP *Constraint Programming* discutiu a necessidade de uma linguagem unificada para seus modelos e pesquisas

➳ Inicialmente a linguagem ZINC foi criada pelo NICTA, Universidade de Melbourne e Universidade de Monash. Tudo na Austrália!

Histórico

➸ Em 2006 a comunidade de CP *Constraint Programming* discutiu a necessidade de uma linguagem unificada para seus modelos e pesquisas

➸ Inicialmente a linguagem ZINC foi criada pelo NICTA, Universidade de Melbourne e Universidade de Monash. Tudo na Austrália!

➸ O MINZINC é um sub-conjunto do ZINC

Histórico

➽ Em 2006 a comunidade de CP *Constraint Programming* discutiu a necessidade de uma linguagem unificada para seus modelos e pesquisas

➽ Inicialmente a linguagem ZINC foi criada pelo NICTA, Universidade de Melbourne e Universidade de Monash. Tudo na Austrália!

➽ O MINZINC é um sub-conjunto do ZINC

➽ Linguagem de modelagem (vários conceitos da lógica)

Histórico

- Em 2006 a comunidade de CP *Constraint Programming* discutiu a necessidade de uma linguagem unificada para seus modelos e pesquisas
- Inicialmente a linguagem ZINC foi criada pelo NICTA, Universidade de Melbourne e Universidade de Monash. Tudo na Austrália!
- O MINZINC é um sub-conjunto do ZINC
- Linguagem de modelagem (vários conceitos da lógica)
- Minizinc é compilado para o FlatZinc – cujo código é traduzido há vários outros *solvers*

Propósitos

➽ Objetivo: resolver problemas de otimização combinatória e PSR (Problemas de Satisfação de Restrições)

➽ O objetivo é descrever o problema: **declarar** no lugar de especificar o que o programa deve fazer

➽ Paradigma de programação <u>imperativo</u>: **como** deve ser calculado !

➽ Paradigma de programação <u>declarativo</u>: **o que** deve ser calculado!

Motivação

O que é um problema combinatório?

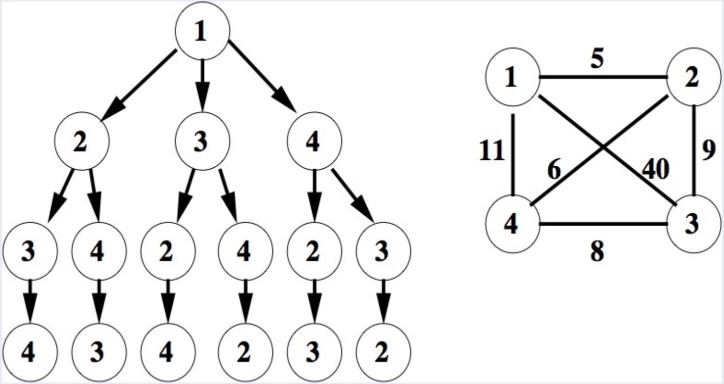


Figura: Problema da sequência de visitas

A complexidade nas coisas simples!

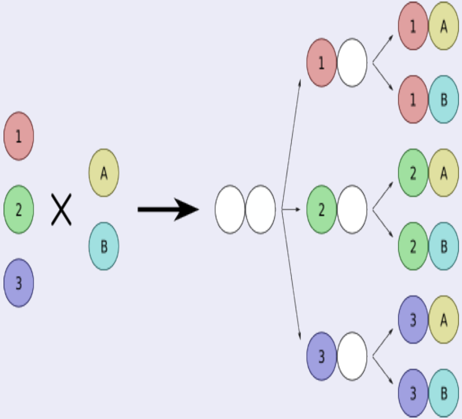


Figura: Contando combinações das variáveis: `X` e `Y`

Introdução

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

Introdução

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

➽ Linguagem $\Rightarrow$ construir modelos $\Rightarrow$ problemas reais

Introdução

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
- Linguagem $\Rightarrow$ construir modelos $\Rightarrow$ problemas reais
- **Modelos** $\Leftrightarrow$ computáveis!

Introdução

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
- Linguagem $\Rightarrow$ construir modelos $\Rightarrow$ problemas reais
- **Modelos** $\Leftrightarrow$ computáveis!
- Visão lógica: insatisfatível (sem respostas) ou consistente

Resumindo alguns livros e *solvers*

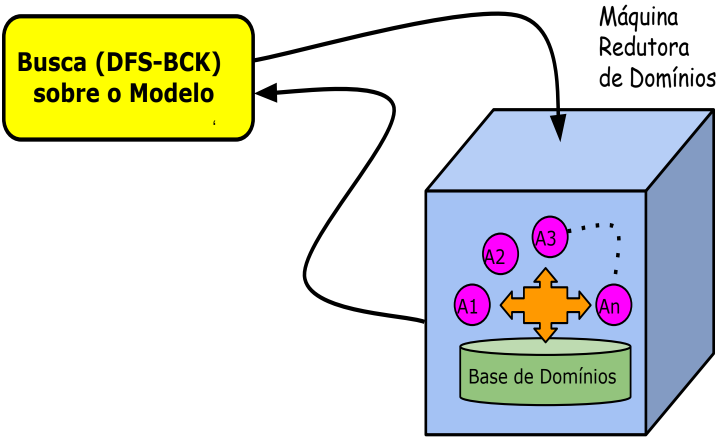


Figura: Ciclo entre a efetiva busca e a poda, na propagação das restrições

Características do MiniZinc:

- Modelagem: imediata à abordagem matemática existente

Características do MiniZinc:

- Modelagem: imediata à abordagem matemática existente
- Para isto, *MUITOS* recursos: operadores booleanos, aritméticos, constantes, variáveis, etc

Características do MiniZinc:

- Modelagem: imediata à abordagem matemática existente
- Para isto, *MUITOS* recursos: operadores booleanos, aritméticos, constantes, variáveis, etc
- **Fortemente tipada**

Características do MiniZinc:

- Modelagem: imediata à abordagem matemática existente
- Para isto, *MUITOS* recursos: operadores booleanos, aritméticos, constantes, variáveis, etc
- **Fortemente tipada**
- *Dois* tipos de dados: <u>constantes</u> e <u>variáveis</u>

Continuando as características:

- Constantes: são valores fixos – são conhecidos como **parâmetros**

Continuando as características:

- Constantes: são valores fixos – são conhecidos como **parâmetros**
- Variáveis: assumem valores sobre um domínio (aqui é o ponto)

Continuando as características:

- Constantes: são valores fixos – são conhecidos como **parâmetros**
- Variáveis: assumem valores sobre um domínio (aqui é o ponto)
- Logo: restringir estes domínios apenas para valores admissíveis

Continuando as características:

- Constantes: são valores fixos – são conhecidos como **parâmetros**
- Variáveis: assumem valores sobre um domínio (aqui é o ponto)
- Logo: restringir estes domínios apenas para valores admissíveis
- Mas há muitos tipos de dados: *int*, *bool*, *real*, *arrays*, *sets*, etc

Continuando as características:

- Constantes: são valores fixos – são conhecidos como **parâmetros**
- Variáveis: assumem valores sobre um domínio (aqui é o ponto)
- Logo: restringir estes domínios apenas para valores admissíveis
- Mas há muitos tipos de dados: *int*, *bool*, *real*, *arrays*, *sets*, etc
- Diferentemente da tipagem dinâmica .... aqui não existe!

Instalar e usar:

Tem evoluído muito nestes últimos anos:

1. **Tudo tem sido simplificado**
2. **Download e detalhes: http://www.minizinc.org/**
3. Basicamente: baixar o arquivo da arquitetura desejada, instalar, acertar variáveis de ambiente, *path*, e usar como:

Instalar e usar:

Tem evoluído muito nestes últimos anos:

1. **Tudo tem sido simplificado**
2. **Download e detalhes: http://www.minizinc.org/**
3. Basicamente: baixar o arquivo da arquitetura desejada, instalar, acertar variáveis de ambiente, *path*, e usar como:
   - Modo console (ou linha de comando) ou
   - Interface IDE

Resumindo

➽ Modo console: `mzn2doc`, `mzn2fzn`, `mzn-g12fd`, `mzn-g12lazy`, `mzn-g12mip`, `mzn-gecode`, ...

1. Edite o programa em um editor ASCII
2. Para compilar e executar:
   mzn-xxxx nome-do-programa.mzn ou escolher um outro *solver*
3. Exemplo como todas soluções:
   mzn-g12fd –all_solutions nome-do-programa.mzn
4. Detalhes e opções: `mzn-g12fd -help`

Resumindo

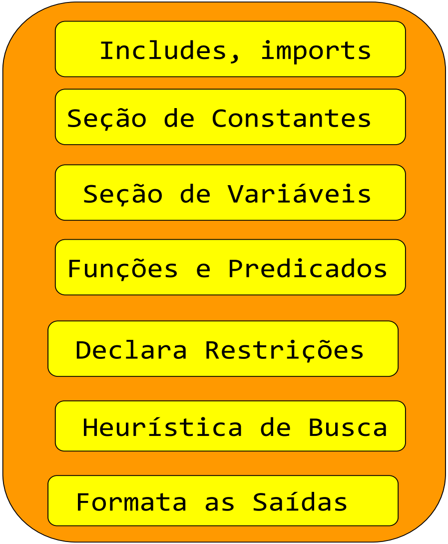
➸ Modo console: `mzn2doc`, `mzn2fzn`, `mzn-g12fd`, `mzn-g12lazy`, `mzn-g12mip`, `mzn-gecode`, `...`

1. Edite o programa em um editor ASCII
2. Para compilar e executar:
   mzn-xxxx nome-do-programa.mzn ou escolher um outro *solver*
3. Exemplo como todas soluções:
   mzn-g12fd –all_solutions nome-do-programa.mzn
4. Detalhes e opções: `mzn-g12fd -help`

➸ Modo IDE: minizinc_IDE ou minizincIDE

➸ Na IDE dá para editar e alterar configurações

Estrutura de um Modelo



Exemplo: um Espaço de Estado (EE)

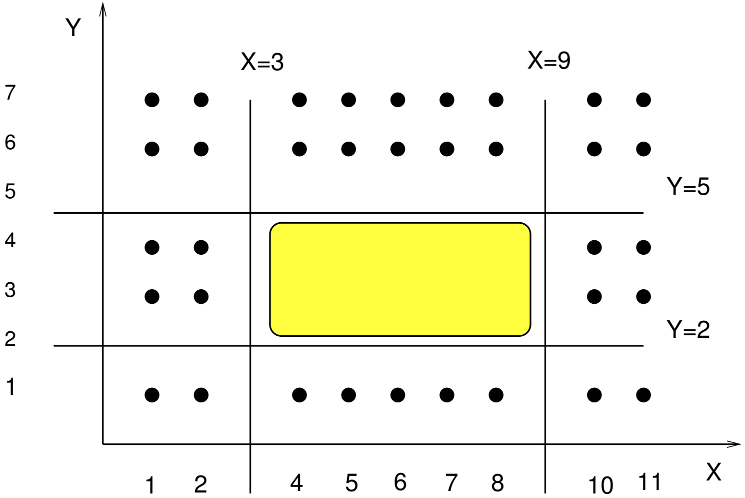


Figura: Obter os pontos do interior do retângulo

Exemplo

```
%% Declara constantes
int: UM = 1; int: DOIS = 2; int:  CINCO = 5;
%% Declara variaveis
var UM  .. 11 : X; %% segue o dominio 1..11
var UM  .. 7 : Y;

%% As restricoes
constraint
   Y >  DOIS  /\    Y <  CINCO ;

constraint
   X >  3    /\      X <  9 ;

%%% A busca : MUITAS OPCOES ....
solve::int_search([X,Y],input_order,indomain_min,complete) satisfy;
%% SAIDAS
output ["   X: ", show(X), "      Y: ", show(Y), "\n"];
```

Saída

```
$ mzn-g12fd -a xerek-ygor.mzn
   X: 4     Y: 3
----------
   X: 4     Y: 4
----------
   X: 5     Y: 3
----------
   X: 5     Y: 4
----------
   X: 6     Y: 3
----------
   X: 6     Y: 4
----------
   X: 7     Y: 3
----------
   X: 7     Y: 4
----------
   X: 8     Y: 3
----------
   X: 8     Y: 4
----------
```

Parâmetros e Variáveis

Existem basicamente dois tipos de variáveis em Minizinc:

Parâmetros e Variáveis

Existem basicamente dois tipos de variáveis em Minizinc:

Parâmetros: *quase igual* às váriaveis de linguagens de programação comuns. Entretanto, só é permitido atribuir um valor a um parâmetro uma única vez.

Parâmetros e Variáveis

Existem basicamente dois tipos de variáveis em Minizinc:

Parâmetros: *quase igual* às váriaveis de linguagens de programação comuns. Entretanto, só é permitido atribuir um valor a um parâmetro uma única vez.

Variáveis de Decisão: mais próximo ao conceito de incógnitas da matemática. O valor de uma váriavel de decisão é escolhido pelo Minizinc para atender todas as restrições estabelecidas.

Parâmetros e Variáveis

Existem basicamente dois tipos de variáveis em Minizinc:

Parâmetros: *quase igual* às váriaveis de linguagens de programação comuns. Entretanto, só é permitido atribuir um valor a um parâmetro uma única vez.

Variáveis de Decisão: mais próximo ao conceito de incógnitas da matemática. O valor de uma váriavel de decisão é escolhido pelo Minizinc para atender todas as restrições estabelecidas.

Variáveis de Restrição: similar a anterior, exceto que o domínio é específico as respostas desejadas do problema

Parâmetros e Variáveis

Existem basicamente dois tipos de variáveis em Minizinc:

Parâmetros: *quase igual* às váriaveis de linguagens de programação comuns. Entretanto, só é permitido atribuir um valor a um parâmetro uma única vez.

Variáveis de Decisão: mais próximo ao conceito de incógnitas da matemática. O valor de uma váriavel de decisão é escolhido pelo Minizinc para atender todas as restrições estabelecidas.

Variáveis de Restrição: similar a anterior, exceto que o domínio é específico as respostas desejadas do problema

Variáveis de Restrição: estas são *descobertas* dentro de um domínio de valores sob um conjunto de restrições que é o **modelo a ser computado**!

Exemplos de Variáveis

Exemplo de Parâmetro (*variável fixa*) em MINIZINC

```
int: parametro = 5;
```

Exemplo de Variável em MINIZINC

```
var 1..15: variavel;
```

Constraints (Restrições)

Restrições podem ser equações ou desigualdades sobre as váriaveis de decisão, de forma a restringir os possíveis valores que estas podem receber.

Constraints (Restrições)

Restrições podem ser equações ou desigualdades sobre as váriaveis de decisão, de forma a restringir os possíveis valores que estas podem receber.

Exemplos de Restrições

```
constraint x > 2;

constraint 3*y - x <= 17;

constraint x != y;

constraint x = 2*z;
```

Alguns Operadores Lógicos

Operadores Lógicos

Os operadores lógicos (*and*, *or*, *not*), que existem na maioria das linguagens de programação, também podem ser utilizados em MINIZINC nas restrições.

Alguns Operadores Lógicos

Operadores Lógicos

Os operadores lógicos (*and*, *or*, *not*), que existem na maioria das linguagens de programação, também podem ser utilizados em MINIZINC nas restrições.

Exemplo de Utilização (and)

```
var bool : p;
var bool : q;
constraint
    (p /\ q) == true;

solve satisfy;

output [show(p), "      ",show(q)];
```

Exemplo de Utilização (*or*)

```
constraint (p \/ q) = false;
```

Exemplo de Utilização (*not*)

```
constraint (not)p = true;
```

Quermesse da Nossa Escola

Exemplo

A escola local fará uma festa e esta precisa que façamos bolos para vender. Sabemos como fazer dois tipos de bolos. Eis a receita de cada um deles:

Quermesse da Nossa Escola

Exemplo

A escola local fará uma festa e esta precisa que façamos bolos para vender. Sabemos como fazer dois tipos de bolos. Eis a receita de cada um deles:

Bolo de Banana	Bolo de Chocolate
- 250g de farinha	- 200g de farinha
- 2 bananas	- 75g de cacau
- 75g de açúcar	- 150g de açúcar
- 100g de manteiga	- 150g de manteiga

Tabela: Insumos de cada bolo

Continuando o enunciado ...

O preço de venda de um Bolo de Chocolate é de R$4,50 e de um Bolo de Banana é de R$4,00. Temos 4kg de farinha, 6 bananas, 2kg de açúcar, 500g de manteiga e 500g de cacau. Qual a quantidade de cada bolo que deve ser feita para maximizar o lucro das vendas para a escola?

Comentários Gerais da Solução

1. Basicamente temos que encontrar: N_1 (número de bolos de Chocolate) e N_2 (número de bolos de Banana);

Comentários Gerais da Solução

1. Basicamente temos que encontrar: N_1 (número de bolos de Chocolate) e N_2 (número de bolos de Banana);
2. Tendo os valores N_1 e N_2, sabemos o nosso lucro máximo, dado o valor por bolo vendido;

Comentários Gerais da Solução

1. Basicamente temos que encontrar: N_1 (número de bolos de Chocolate) e N_2 (número de bolos de Banana);
2. Tendo os valores N_1 e N_2, sabemos o nosso lucro máximo, dado o valor por bolo vendido;
3. Assim a equação a ser maximizada é: $4500.N_1 + 4000.N_2 = Lucro$

Comentários Gerais da Solução

1. Basicamente temos que encontrar: N_1 (número de bolos de Chocolate) e N_2 (número de bolos de Banana);
2. Tendo os valores N_1 e N_2, sabemos o nosso lucro máximo, dado o valor por bolo vendido;
3. Assim a equação a ser maximizada é: $4500.N_1 + 4000.N_2 = Lucro$
4. Sabe-se que **UM** bolo necessita de quantidades de insumos dado na tabela 1

Comentários Gerais da Solução

1. Basicamente temos que encontrar: N_1 (número de bolos de Chocolate) e N_2 (número de bolos de Banana);
2. Tendo os valores N_1 e N_2, sabemos o nosso lucro máximo, dado o valor por bolo vendido;
3. Assim a equação a ser maximizada é: $4500.N_1 + 4000.N_2 = Lucro$
4. Sabe-se que **UM** bolo necessita de quantidades de insumos dado na tabela 1
5. Logo, se são N bolos por insumos e respeitando a disponibilidade de cada um, as restrições para ambos os bolos são do tipo: $N_1.qt_{manteiga_{chocolate}} + N_2.qt_{manteiga_{banana}} \leq Manteiga_{disponivel}$

Comentários Gerais da Solução

1. Basicamente temos que encontrar: N_1 (número de bolos de Chocolate) e N_2 (número de bolos de Banana);
2. Tendo os valores N_1 e N_2, sabemos o nosso lucro máximo, dado o valor por bolo vendido;
3. Assim a equação a ser maximizada é: $4500.N_1 + 4000.N_2 = Lucro$
4. Sabe-se que **UM** bolo necessita de quantidades de insumos dado na tabela 1
5. Logo, se são N bolos por insumos e respeitando a disponibilidade de cada um, as restrições para ambos os bolos são do tipo: $N_1.qt_{manteiga_{chocolate}} + N_2.qt_{manteiga_{banana}} \leq Manteiga_{disponivel}$
6. E estes valores são tomados da tabela 1.

Uma tabela *conhecida* tipo:

	farinha	cacau	bananas	açucar	manteiga	
N_1 (Choco)	200	75	–	150	150	
N_2 (Banana)	250	–	2	75	100	
Disponível	4000	500	6	2000	500	

Código desta solução:

```
var 0..100: bc; %Bolo de chocolate: N1
var 0..100: bb; %Bolo de banana: N2

constraint 250*bb + 200*bc <= 4000;
constraint 2*bb <= 6;
constraint 75*bb + 150*bc <= 2000;
constraint 100*bb + 150*bc <= 500;
constraint 75*bc <= 500;

solve maximize (4500*bc + 4000*bb);

output[" Choc = ", show(bc), "\t Ban = ",show(bb)];
```

Saída

```
Compiling bolos.mzn
Running bolos.mzn
 Choc = 0  Ban = 0
----------
 Choc = 1  Ban = 0
----------
 Choc = 2  Ban = 0
----------
 Choc = 3  Ban = 0
----------
 Choc = 2  Ban = 2
----------
==========
Finished in 36msec
```

Teoria dos Conjuntos

```
set of int: B = {1,2,3};
% OU set of int: B = 1 .. 3;
set of int: A = {4,5};

var set of   1 .. 5 : var_uniao;
var set of   1 .. 5 : var_inters ;

constraint
  var_uniao = B union A;

constraint
  var_inters = B intersect A;

solve satisfy;

output
     ["VAR_Uniao = " , show(var_uniao),"\n",
      "VAR_Inters = " , show(var_inters),"\n"];
```

Construindo Funções

```
int: n = 3;
var int: z1;
var int: z2;

function var int: pot_3_F(var int: n) = n*n*n ;

predicate pot_3_P(int: n, var int: res) =
          res =  n*n*n ;

constraint
     z1 = pot_3_F(n) ;

constraint
     pot_3_P(n,z2) ;

solve satisfy;

output ["n: ", show(n),"\n", "z1: ", show(z1), "\n",
        "z2: ", show(z2), "\n"];
```

Saída

Finished in 400msec

Compiling funcao_01.mzn

Running funcao_01.mzn

n: 3

z1: 27

z2: 27

———-

Finished in 54msec

Funções

```
int : X = 5 ; %%  constantes
int : Y = 6;
var bool : var_bool_01;
var bool : var_bool_02;

%%% Temos if-then-else-endif
function var bool :  testa_paridade(int : N) =
     if( (N mod 2) == 0)
        then
          true
        else
          false
      endif;

constraint
   var_bool_01 == testa_paridade(X);

constraint
   var_bool_02 == testa_paridade(Y);

/* OR  var_bool_01 == ((x mod 2) == 0);
       var_bool_02 == ((y mod 2) == 0);   */

solve satisfy;

```

Continuando ...

```
........................
solve satisfy;

output
    ["  CTE_X = ", show(X), "  CTE_Y = ", show(Y), "\n",
     "  VAR_B01 = ", show( var_bool_01 ),
     "  VAR_B02 = ", show( var_bool_02 )  ] ;
```

Continuando ...

```
.........................
solve satisfy;

output
    ["  CTE_X = ", show(X), "  CTE_Y = ", show(Y), "\n",
     "  VAR_B01 = ", show( var_bool_01 ),
     "  VAR_B02 = ", show( var_bool_02 )  ] ;
```

Saída:

```
$ mzn-g12fd -a minizinc/bool_function.mzn
  CTE_X = 5  CTE_Y = 6
  VAR_B01 = false  VAR_B02 = true
----------
==========
```

Uso na Lógica Proposicional

```
var bool : x;
var bool : y;
var bool : Phi01;
var bool : Phi02;

constraint                  %% MODUS PONENS
      ((x /\
      (x -> y)) -> y)
      <-> Phi01 ;

constraint                              %% MODUS TOLLENS
      ((not y /\
      (x -> y)) -> not x)
      <-> Phi02 ;

solve satisfy;

output
[" X: "++show(x)++"   Y: "++ show(y) ++"  MP:Phi01: "++ show(Phi01)++
[" X: "++show(x)++"   Y: "++ show(y) ++"  MT:Phi02: "++ show(Phi02)++
```

Saída:

```
$ mzn-g12fd -a minizinc/interp_log_MP.mzn
 X: false   Y: false  MP:Phi01: true
 X: false   Y: false  MT:Phi02: true
----------
 X: true   Y: false  MP:Phi01: true
 X: true   Y: false  MT:Phi02: true
----------
 X: false   Y: true  MP:Phi01: true
 X: false   Y: true  MT:Phi02: true
----------
 X: true   Y: true  MP:Phi01: true
 X: true   Y: true  MT:Phi02: true
----------
==========
```

Interpretação na Lógica de Primeira-Ordem

Sejam as FPO abaixo:

- Exemplo 01: $\forall x \; \exists y \; (y < x)$
- Exemplo 02: $\exists x \; \forall y \; (x < y)$
- Exemplo 03: $\forall x \; \exists y \; (x^2 == y)$
- Exemplo 04: $\exists x \; \forall y \; (x^2 != y)$
- Avalie a validade para os domínios: $D_x = \{2, 3, 4\}$ e $D_y = \{3, 4, 5\}$

Interpretação na Lógica de Primeira-Ordem

```
%%Declarando dominio das  variaveis

set of int: X = {2, 3, 4};
set of int: Y = {3, 4, 5};

function bool: exemplo_01(set of int: x, set of int: y) =
        (forall (i in x) (exists (j in y) (j < i)));

function bool: exemplo_02(set of int: x, set of int: y) =
        exists (i in x) (forall (j in y) (i < j));

function bool: exemplo_03(set of int: x, set of int: y) =
        forall (i in x) (exists (j in y) (pow(i,2) == j));

function bool: exemplo_04(set of int: x, set of int: y) =
        exists (i in x) (forall (j in y) (pow(i,2) != j));

solve satisfy;

output["\n Exemplo 01: "++ show(exemplo_01(X,Y))++
       "\n Exemplo 02: "++ show(exemplo_02(X,Y))++
       "\n Exemplo 03: "++ show(exemplo_03(X,Y))++
       "\n Exemplo 04: "++ show(exemplo_04(X,Y))];
```

Saída:

```
$ mzn-g12fd -a minizinc/interp_fol_set.mzn

 Exemplo 01: false
 Exemplo 02: true
 Exemplo 03: false
 Exemplo 04: true
----------
==========
```

Vetores (*Arrays*)

Vetores 1-D

- Seja `int :  n = 7;`
- `array[1..n] of int :  vetor01; (constante)`
- `array[1..n] of {0,1,2,3} :  vetor02; (constante)`
- `array[1..n] of var { 0,1 } :  vetor03;`

Vetor 1-D, variáveis locais e escopo

```
int: n = 7; %% total de elementos
int: m = 4; %% m itens a serem selecionados

array[1..n] of var {0,1} : x_decision;

%% OK e direto via sum( i in 1..n ) (vetor_1d[i]);
function var int: sum_array_1d(array[1..n] of var int: vetor_1d) =
 let{
      array[1..n] of var int : temp;
      constraint                          %%%% C_1
      temp[1] == vetor_1d[1];
      constraint                          %%%% C_2
      forall(i in 2..n)
        ( temp[i] == temp[i-1] + vetor_1d[i] );
      } in temp[n] %%% Valor acumulado aqui
    ;

%%% constraint  m == sum( i in 1..n ) (x_decision[i]);

constraint
   m == sum_array_1d( x_decision );

solve satisfy;
output [" x_decision: " ++ show(x_decision) ];
%       " Lower Bound: ", show(lb_array(x_decision)), "\n",
```

Saída:

```
mzn-g12fd -a minizinc/function_sum_vetor_1D.mzn
x_decision: [0, 0, 0, 1, 1, 1, 1]
----------
 x_decision: [0, 0, 1, 0, 1, 1, 1]
----------
 x_decision: [1, 0, 0, 0, 1, 1, 1]
 ..............................................
 x_decision: [1, 1, 0, 1, 1, 0, 0]
----------
 x_decision: [1, 1, 1, 0, 1, 0, 0]
----------
 x_decision: [1, 1, 1, 1, 0, 0, 0]
----------
==========
```

Exemplos de 1D ..... TODO

Vetores 2-D

```
/* EXERCICIO
Dado um vetor bi-dimensional, crie uma funcao que calcule e retorne a soma de todos
elementos desta matriz. Ao fazer esta funcao, faca uma que imprima os valores da
matriz. Teste-a na secao do output do Minizinc;
*/

int: Lin = 4;
int: Col = 10;

array[1 .. Lin, 1 .. Col] of int: G;

G = [|1,2,3,4,5,6,7,8,9,10,
      |1,2,3,4,5,6,7,8,9,10,
      |1,2,3,4,5,6,7,8,9,10,
      |1,2,3,4,5,6,7,8,9,10
      |];

var int : final_result;

function var int: matrix_sum(array[1.. Lin,1 .. Col] of int: matrix) =
    let{
      var int : temp;
      array[1..Lin] of var int: partial_line;
      constraint
      forall( i in 1..Lin )
       (partial_line[i] == sum(j in 1 .. Col) (matrix[i,j])
       )
      /\
      temp == sum(partial_line);
    } in temp   %%% AQUI O RETORNO DA FUNCAO
;

constraint
   final_result == matrix_sum(G);
```

Mais exemplos: vetores e matrizes

Resolução

- Uni-dimensional
- Bi-dimensional (tem nomes especiais)
- n-ários ... volta há um padrão *default* de uso
- Falta um exemplo simples de uma dimensão: fazer em sala

Vetores e Matrizes

Quadrado Mágico

Um quadrado mágico é uma matriz NXN onde os somatórios das linhas, colunas e diagonais (principal e secundária) são todos iguais a um valor Σ. Além disso, os elementos da matriz devem ser diferentes entre si e com valores entre 1 e N. Em MINIZINC, faça um programa que, dado o valor da soma Σ, encontre um quadrado mágico de ordem 4.

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

Do Wikipedia

The constant that is the sum of every row, column and diagonal is called the *magic constant* or magic sum, M. Every normal magic square has a unique constant determined solely by the value of n, which can be calculated using this formula:

$$M = \frac{n(n^2+1)}{2}$$

For example, if $n = 3$, the formula says $M = 3(3^2+1)/2$, which simplifies to 15. For normal magic squares of order $n = 3$, 4, 5, 6, 7, and 8, the magic constants are, respectively: 15, 34, 65, 111, 175, and 260.
Sim, outro detalhe, válido para: $n > 2$

Resolução

acertar o arquivo fonte ....

```
int: soma;

array[0..3,0..3] of var 1..16: mat;

constraint forall(i in 0..3)
        (mat[i,0] + mat[i,1] + mat[i,2] + mat[i,3] = soma);
constraint forall(j in 0..3)
        (mat[0,j] + mat[1,j] + mat[2,j] + mat[3,j] = soma);
constraint mat[0,0] + mat[1,1] + mat[2,2] + mat[3,3] = soma;
constraint mat[0,3] + mat[1,2] + mat[2,1] + mat[3,0] = soma;
constraint forall(i in 0..3, j in 0..3, k in i..3, l in j..3)
        (if (i!= k \/ j != l) then mat[i,j] != mat[k,l] else true endi
constraint forall(i in 0..3, j in 0..3)(mat[i,j] <= 16);

solve satisfy;

output[show_int(2,mat[i,j]) ++
        if j==3 then "\n" else " " endif |
        i in 0..3, j in 0..3];
```

Grafos

Figura: Regiões da Itália

Modelando o Mapa

```
%%  Pgm origem  coloracao_vertices01.mzn
 E = [|0,1,0,0,0,0,0,0
      |1,0,1,1,1,0,0,0
      |0,1,0,1,0,0,0,0
      |0,1,1,0,1,1,0,0
      |0,1,0,1,0,1,1,0
      |0,0,0,1,1,0,1,1
      |0,0,0,0,1,1,0,0
      |0,0,0,0,0,1,0,0|];
%% CIDADES ....
% 1 Friuli Venezia Giulia
% 2 Veneto
% 3 Trentino Alto Adige
% 4 Lombardy
% 5 Emilia-Romagna
% 6 Piedmont
% 7 Liguria
% 8 Aosta Valley
```

As Restrições

```
int: n=8;
int: c=4;
array [1..n,1..n] of int: E;    %% regioes
array [1..n] of var 1..c: Col; %% cores

constraint
  forall (i in 1..n, j in i+1..n)
     (if E[i,j] = 1 then Col[i] != Col[j] else true endif);

solve satisfy;

output [show(Col)];
```

Coloração de Mapas – by – HAKAN

```
%     coloracao_vertices02.mzn
% between two countries
%
graph =
array2d(1..num_nodes, 1..2, [
 3, 1,
 3, 6,
 3, 4,
 6, 4,
 6, 1,
 1, 5,
 1, 4,
 4, 5,
 4, 2
]);
```

Modelando o Mapa

```
%          1                2                  3               4                5
% {"Belgium", "Denmark", "France", "Germany", "Netherlands", "I
set of 1..6: countries = 1..6 ; % the countries
int: n = card(countries);       % number of countries
set of 1..n: colors = 1..n;     % the colors

int: num_nodes = 9;             % number of nodes
array[1..num_nodes,1..2] of int: graph;

% x: what color
array[countries] of var 1..n: x;

% minimize the number of colors .... MINIMIZA AQUI ....
solve minimize numColors;
```

Finalmente a restrição do problema

```
% Uma restricao
constraint
   % No adjacent countries can have the same color
   forall(i in 1..num_nodes) (
      x[graph[i,1]] != x[graph[i,2]]
   )
   /\ % upper limit on the number of colors
   forall(i in countries) (
     x[i] <= numColors
   )
%  /\ numColors <= 4
;
```

Ilustrando a combinatória

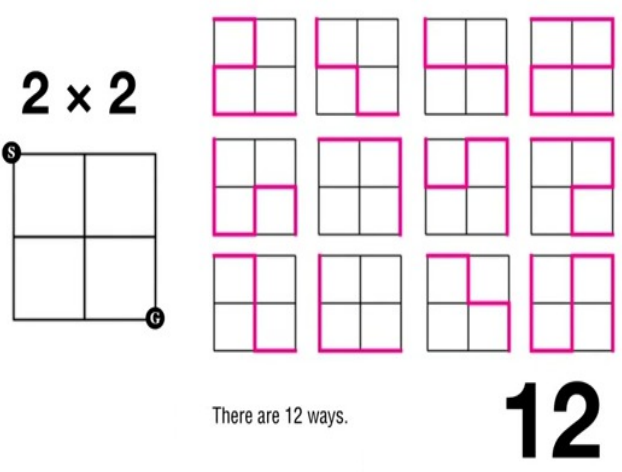


Figura: Problema de rotas alternativas

Modelando este problema

```
int : src = 1; int : dst = 9; int : n = 9;
array[1..n, 1..n] of 0..1 : G = [|
  1, 1, 0, 1, 0, 0, 0, 0, 0 |
  1, 1, 1, 0, 1, 0, 0, 0, 0 |
  0, 1, 1, 0, 0, 1, 0, 0, 0 |
  1, 0, 0, 1, 1, 0, 1, 0, 0 |
  0, 1, 0, 1, 1, 1, 0, 1, 0 |
  0, 0, 1, 0, 1, 1, 0, 0, 1 |
  0, 0, 0, 1, 0, 0, 1, 1, 0 |
  0, 0, 0, 0, 1, 0, 1, 1, 1 |
  0, 0, 0, 0, 0, 1, 0, 1, 1 |];


% Grafo de decisao que representa o resultado (r)
array[1..n, 1..n] of var 0..1 : r;
```

As restrições

```
% aceita somente arcos validos da matriz
constraint
    forall(i,j in 1..n where i != dst)
    (G[i,j]==0 -> r[i,j]==0) /\ r[dst,src]=1;


% O grafo deve ser conservativo : bidirecional
constraint
    forall(i in 1..n)
    (sum([r[j, i] | j in 1..n]) =
     sum([r[i, j] | j in 1..n]));
%% todos que chegam ha um NO .... saem
```

As restrições

```
% um no pode ter no maximo uma aresta no sentido i -> j
constraint
      forall(i in 1..n)
         (sum([r[i, j] | j in 1..n]) < 2);

% deve existir uma aresta de src para algum no
constraint
      exists(i in 1..n)
      (r[src,i] == 1 /\ i != src);

solve satisfy;

% Output do Hakank
output [show(r[i,j]) ++ if j = n then "\n" else " "
             endif | i in 1..n, j in 1..n];
```

Caminho Mínimo

1. **Muitas estratégias de implementação!!!**
2. Discutido os códigos abaixo

```

Ver codigos:
min_path01.mzn  min_path02.mzn  min_path03.mzn
%% comentado no codigo
```

Conclusões

➳ Exemplos de códigos avançados:
`https://github.com/hakank/hakank/tree/master/minizinc`