Tutorial sobre MiniZinc

Claudio Cesar de Sá e outros

31 de Julho de 2016

Conteúdo

1

Tutorial sobre MiniZinc

1.1 Contexto

1.2 Motivação

1.2.1 Definições Iniciais

O MiniZincé uma *linguagem de modelagem* de alto-nível, destinada a descrever modelos matemáticos para problemas de satisfação de restrições e otimização combinatória. Uma vez construído um modelo, uma especificação em MiniZinc, esta precisa ser processada por algum *backend*. Ou seja, algum solucionador ou resolvedor, comumente conhecido como *solver*, que processe suas especificações e retorne valores consistentes, ou até mesmo ótimos.

Assim, os *solvers* desempenham papel fundamental em problemas de combinatória. Existem vários *solvers*, de vários propósitos, e que operam em domínios de valores diversos.

Deste modo, o MiniZincé uma linguagem de modelos que suporta uma codificação única, contudo, com chamada há vários tipos destes *solvers*. Logo, MiniZincfoi criado com o objetivo de unificar em uma única linguagem, a chamada aos mais diferentes *solvers*.

Deste modo, uma única implementação em MiniZincpode ser comparada com diversos *solvers*, permitindo que estes sejam comparados com imparcialidade entre si. Este fato, permite que um dado modelo, bem ou mal modelado, seja único sob os diversos *solvers* em *backend*.

Esta situação é ilustrada na figura ....

1.2.2 Linguagens e Bibliotecas

Para se encaminhar a resolução de problemas de otimização discreta, a codificação do mesmo pode ser feita basicamente sob duas visões de uso:

- Linguagens de hospedeiras
- Linguagens orientadas a modelo

A linguagens de hospedeiras disponibilizam de toda sua sintaxe e uma biblioteca específica que suporte o paradigma da Programação por Restrições (PR) ou ainda que possa utilizar bibliotecas prontas de outros resolvedores (*solvers*).

A principal característica para estas linguagens é a sua flexibilidade na construção de detalhes do modelo. Aspectos como a depuração, controle das buscas, criar novos artefatos para busca, etc, tornam este grupo atrativo para um desenvolvimento de alto-desempenho.

Como exemplo para este grupo segue as seguintes linguagens e suas respectivas bibliotecas:

Picat apresenta uma biblioteca própria para PR, e pode chamar outros *solvers*;

GECODE é uma biblioteca própria para PR escrita em C++, e utiliza toda linguagem C++ como ambiente de programação;

ECLiPSe é um ambiente completo para PR estendendo a linguagem Prolog. O código do ECLIPSE é aberto, e o mesmo é mantido pela empresa CISCO;

SICTUS é proprietário e para PR estende a linguagem Prolog.

JACOP é uma biblioteca que estende a a linguagem Java. Para uso acadêmico é livre, para fins indústriais proprietário.

Para o grupo de linguagens orientadas a modelos, o foco é a *descrição dos modelos* e o uso de *solvers* diversos.

CPLEX produto da IBM, originário do ILOG;

MiniZinc um subconjunto da linguagem Zinc, e que utiliza o *solver* FlatZinc;

Gurobi ;

Gurobi ;

1.3 Apresentando o MiniZinc

Conforme [1], 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 [1]. A simplicidade se encontra principalmente na sintaxe, e a expressividade é dada por permitir modelar diversos tipos de problemas por meio dos recursos disponibilizados.

1.4 Operacionalidade entre *solvers* e o MiniZinc

1.5 Elementos da Linguagem

1.5.1 Um Paradigma Computacional

Incluir aqui partes do artigo já escrito

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

- 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!

1.5.2 Estrutura de um Programa em MiniZinc

Faltando uma figura

1.5.3 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 ....

1.5.4 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 sob o *modelo computado*. Esta é a idéia da PR.

1.5.5 Restrições (Declarações Lógicas)

Faltando ....

1.5.6 Satisfatibilidade e Otimização

Faltando ....

1.5.7 Saídas

Faltando ....

1.5.8 Instalação e Uso

Imediato após a versão 2.0.12

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

Melhorar isto acima

1.6 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 1.1.

```
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 1.1: Exemplo de um modelo em MiniZinc

Este é um modelo para o problema das n-rainhas, apresentado na Subseção **??**. 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 1.2, 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 **??**.

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 **??**. 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 1.2 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 1.2: 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.

1.7 Conclusões Parciais

Faltando .... e melhorar a apresentacao acima

Bibliografia

[1] 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.