UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
PARADIGMAS DE PROGRAMAÇÃO

Trabalho III - Programação Lógica - Prolog Wolkenkratzer

Gabriel Baiocchi de Sant'Anna
Teo Haeser Gallarza

PROFESSOR
Maicon Zatelli

Florianópolis
Outubro de 2019

Programação de Restrições

A programação de restrições (*Constraint Programming*) é um tipo de paradigma de programação utilizado para resolver problemas utilizando restrições definidas através de relações entre as entradas e saídas do programa. Diferente do paradigma imperativo, a programação de restrições não descreve os passos necessários para alcançar um resultado, mas dita as propriedades das possíveis soluções; tornando-se assim muito mais genérica. A compatibilidade da programação de restrições com o paradigma declarativo é evidente e muitas linguagens incluem bibliotecas ou até mesmo mecanismos nativos para possibilitar o uso dessas técnicas.

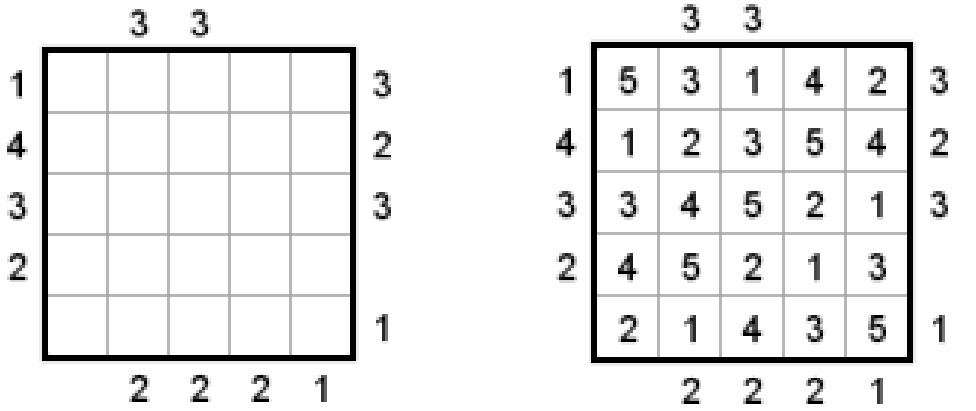
Em Prolog, a programação de restrições é disponibilizada através da biblioteca *clpfd* (*Constraint Logic Programming over Finite Domains*) da implementação SWI-Prolog. A biblioteca é usada para propagar restrições no domínio dos números inteiros através de relações aritméticas como igualdades e inequações e é especialmente útil para resolver problemas combinatórios quando atrelada ao *backtracking* automático da linguagem Prolog. Vale ressaltar que os predicados da *clpfd* são muito mais poderosos e sofisticados do que os operadores nativos em Prolog que operam sobre números inteiros.

```
%% Constraint Logic Programming over Finite Domains
:- use_module(library(clpfd)).

%% exemplo de uso de restrições aritmeticas
% restringe uma lista a sequencia ordenada dos inteiros em [Low, High)
range(Low, High, []) :- High #=< Low.
range(Low, High, [Low|Range]) :-
    High #> Low, Next #= Low + 1, range(Next, High, Range).

%% exemplo de restricao de pertencimento,
% restringe todos os valores de Cells (uma lista) a um certo dominio
Cells ins Min..Max.

%% exemplo de restricao combinatoria
% relaciona os elementos de uma lista de forma que sejam distintos entre si
% isso eh aplicado sobre todas as listas em Rows (uma lista de listas)
maplist(all_distinct, Rows).
```

Neste trabalho utilizamos a programação de restrições para desenvolver um programa que resolve instâncias quaisquer do *puzzle Wolkenkratzer*.

Wolkenkratzer

O jogo Wolkenkratzer, também conhecido como Skyscrapers, é um quebra-cabeça que consiste em um tabuleiro quadrado de lado N com casas que devem ser preenchidas por números de 1 a N – em alguns puzzles são permitidos espaços em branco junto de valores de 1 a N-1 – que não podem ser repetidos entre linhas ou entre colunas (como no famoso Sudoku). Em torno do tabuleiro são colocadas algumas restrições que representam o número de prédios que poderiam ser vistos por um observador naquela posição que olhasse na direção do tabuleiro. Assim, cada número colocado em uma casa do tabuleiro representa um prédio daquela altura que encobre prédios vizinhos de menor altura em cada direção.



(a) Estado inicial do *puzzle*. (b) Solução dessa instância do jogo.

Figura 1 – Exemplo de instância do jogo Wolkenkratzer.

A implementação do *puzzle* em Prolog é simples: basta expressar as regras do quebra-cabeça e a relação das restrições (número de prédios observados a partir das bordas) com o tabuleiro inicialmente livre. Utilizamos uma lista de listas para representar o tabuleiro e uma tupla de listas para representar as restrições em cada uma das quatro bordas, onde um valor 0 é usado para indicar que não há restrição (em branco). Além disso, recebemos um parâmetro para indicar a altura máxima que um prédio pode tomar e se é ou não necessário conferir as diagonais do tabuleiro (algumas instâncias requerem que não hajam repetições nas diagonais).

```
/* solves an instance of the skyscrapers puzzle defined by given constraints
  (0s are ignored), maximum building height and the option to check diagonals */
wolkenkratzer(Board, (Upper, Left, Bottom, Right), Max, DiagCheck) :-
    %% continua ...
```

Primeiramente conferimos o tamanho do tabuleiro (e se as restrições possuem um tamanho compatível com o mesmo). Logo, é possível delimitar o intervalo em que se encontrarão os valores de alturas dos prédios nas células do tabuleiro, o que é feito através da restrição *ins*.

```
    %% ...

    % for a NxN board, each list of constraints has N values
    length(Board, N),
    length(Upper, N),
    length(Left, N),
    length(Bottom, N),
    length(Right, N),

    % cells can be filled with an integer value in the range [Min,Max]
    Min is 1 - (N - Max),
    Min >= 0,
    flatten(Board, Cells),
    Cells ins Min..Max,

    %% ...
```

Em seguida adicionamos a restrição de que cada linha possui elementos distintos, o mesmo valendo também para cada coluna. Obtemos também uma versão invertida das linhas e colunas do tabuleiro, o que será útil para aplicar as restrições de todas as bordas.

```
    %% ...
    % rows dont repeat values and neither do columns
    Rows = Board,
    transpose(Rows, Columns),
    maplist(all_distinct, Rows),
    maplist(all_distinct, Columns),
    maplist(reverse, Rows, ReversedRows),
    maplist(reverse, Columns, ReversedColumns),
    %% ...
```

Quando for necessário, verifica-se também se os elementos nas diagonais são distintos. Note que o metapredicado *maplist* foi muito útil para tornar o código curto e expressivo.

```
    %% ...
    % sometimes, diagonals need to be distinct as well
    diagonal_(DiagCheck, Rows, ReversedRows),
    %% ...

% auxiliary predicate that checks both wolkenkratzer diagonals only if needed
diagonal_(false, _, _).
```

```
diagonal_(true, Rows, ReversedRows):-
    diagonal(Rows, MainDiag), all_distinct(MainDiag),
    diagonal(ReversedRows, SecondDiag), all_distinct(SecondDiag).

% extracts main diagonal from a list of lists
diagonal(Rows, Diag) :-
    length(Rows, N), range(0, N, Indices), maplist(nth0, Indices, Rows, Diag).
```

Assim, basta verificar se o número de prédios observados do ponto de vista de cada posição na borda do tabuleiro respeita a restrição fornecida (se diferente de zero). Os predicados da biblioteca *clpfd* são utilizados para garantir que as soluções do programa levem em conta as regras do jogo, onde o mecanismo de *backtracking* de Prolog tenta fixar variáveis nas células do tabuleiro e falha sempre que os valores neste deixam de respeitar as restrições aplicadas anteriormente.

```
    %% ...

    % check if all constraints are met
    maplist(skyscrapers, Rows, Left),
    maplist(skyscrapers, ReversedRows, Right),
    maplist(skyscrapers, Columns, Upper),
    maplist(skyscrapers, ReversedColumns, Bottom),

    % collapse cell domains into single values
    maplist(label, Board).


% counts the number of visible skyscrapers on a given sequence
skyscrapers(_, 0).
skyscrapers([0|Rest], Constraint) :- skyscrapers(Rest, 0, Constraint, 0).
skyscrapers([First|Rest], Constraint) :-
    First #> 0, skyscrapers(Rest, First, Constraint, 1).

/* checks whether the number of visible skyscrapers in given sequence satisfies
   a constraint, starting with some max height and a previous count */
skyscrapers([], _, Constraint, Constraint).
skyscrapers([First|Rest], Max, Constraint, Seen) :-
    (First #> Max, Num #= Seen + 1, skyscrapers(Rest,First,Constraint,Num));
    (First #=< Max, skyscrapers(Rest,Max,Constraint,Seen)).
```

O código mostrado já é suficiente para obter todas as soluções de instâncias quaisquer de Wolkenkratzer. Entretanto, tabuleiros maiores evidenciam que o mecanismo de busca não é ótimo, com tempos de computação na ordem de dezenas de minutos para obter uma solução mais difícil. Sendo assim, aplicamos algumas regras – observadas ao jogar o jogo – que podem ser aplicadas para reduzir as possibilidades consideradas pelo sistema de tentativa e erro.

```
    %% ...

    % apply pruning
    maplist(pruned(Min,Max), Rows, Left),
    maplist(pruned(Min,Max), ReversedRows, Right),
    maplist(pruned(Min,Max), Columns, Upper),
    maplist(pruned(Min,Max), ReversedColumns, Bottom),

    %% ...
    % check if all constraints are met
    %% ...
```

É fácil perceber que uma restrição com o maior valor permitido implica uma sequência crescente de valores, assim como uma restrição de um único prédio pode ser resolvida de imediato com o prédio mais alto na célula adjacente. Outra regra implementada consiste em encontrar um limitante superior na linha sobre a qual uma restrição se aplica, partindo do princípio de que valores mais altos devem estar mais longe das bordas para que seja possível observar mais prédios no tabuleiro.

```
%% applies pruning laws to further constrain value domains
% zero restrictions are ignored
pruned(_, _, 0).

% when tip is 1 and no parks are allowed, first cell is N
pruned(1, N, [N|_], 1).

% when tip is 1 and parks are allowerd, first cell is either Max or 0
pruned(0, Max, [First|_], 1) :- First #= Max; First #= 0.

% when tip is N, line is [1..N] (no parks)
pruned(1, N, Sequence, N) :- End #= N + 1, range(1, End, Sequence).

% when tip is some K, apply an upper bound to the whole line
pruned(_, N, Line, Tip) :- pruned_(N, Line, Tip).
```

```
pruned_(_, _, 0).
pruned_(_, [], _).
pruned_(N, [First|Rest], Tip) :-
    K is Tip - 1, First #=< N - K, pruned_(N, Rest, K).
```

A adição dessas regras simples trouxe melhorias imediatamente perceptíveis à performance do programa, sendo possível (em alguns casos) inclusive fixar valores no tabuleiro somente através delas. Todavia, após a inclusão desses predicados, o programa passou a fornecer múltiplas soluções iguais à uma mesma consulta.

Entrada e Saída de Dados

O resolvedor encontra-se abstraído no predicado *wolkenkratzer/4*, portanto para utilizar o programa é necessário iniciar o interpretador de Prolog (SWI-Prolog, neste caso) e carregar as definições. Em seguida, basta fornecer os parâmetros que definem uma instância do *puzzle* e o sistema Prolog irá inferir as soluções existentes. Segue a seguir uma consulta exemplificando o uso do programa para o quebra-cabeça ilustrado na Figura 1. Destaca-se que o programa também funciona para tabuleiros parcialmente resolvidos, bastando substituir valores conhecidos diretamente no tabuleiro.

```
%% consulta de entrada
?- Board =  [[_,_,_,_,_],
             [_,_,_,_,_],
             [_,_,_,_,_],
             [_,_,_,_,_],
             [_,_,_,_,_]],
   Board = [A,B,C,D,E],
   wolkenkratzer(
       Board,
       (
           [0,3,3,0,0], % borda superior
           [1,4,3,2,0], % borda esquerda
           [0,2,2,2,1], % borda inferior
           [3,2,3,0,1]  % borda direita
        ),
       5, false % altura maxima 5, sem diagonais
   ).

%% saida fornecida no interpretador
Board = [A,B,C,D,E],
```

```
A = [5, 3, 1, 4, 2],
B = [1, 2, 3, 5, 4],
C = [3, 4, 5, 2, 1],
D = [4, 5, 2, 1, 3],
E = [2, 1, 4, 3, 5] .
```

Vantagens, Desvantagens e Dificuldades

O programa implementado consiste em um quebra-cabeça, tornando evidente as vantagens da programação lógica-declarativa de Prolog com seus mecanismos automáticos de *backtracking* e unificação. Além disso, o problema é naturalmente compatível com a programação de restrições ofertada na biblioteca *clpfd*, visto que consiste, essencialmente, em gerar um tabuleiro preenchido com números inteiros que respeitem certas restrições. Assim, foi possível obter uma implementação boa e razoavelmente eficiente sem muito esforço e com um código curto e compreensível.

A principal dificuldade foi escolher, a cada momento, entre a utilização dos predicados numéricos nativos da linguagem Prolog e os da biblioteca de restrições: onde seriam necessários e em que trechos não fariam diferença alguma. O uso do corte (*!/0*) também foi considerado inicialmente, visando atingir melhor performance; percebeu-se, entretanto, que sua presença impedia a obtenção de múltiplas soluções do quebra-cabeça e em alguns casos fazia com que o programa não funcionasse corretamente.

Paradigma Lógico *vs* Paradigma Funcional

O paradigma lógico mostrou-se especialmente útil para resolver o problema deste trabalho, com diversas construções da linguagem atuando para facilitar o processo de resolução do *puzzle*. Além disso, a depuração do programa declarativo foi simples e intuitiva: tendo a noção de que cada regra torna o conjunto de soluções sempre mais específico, basta comentar temporariamente alguns trechos de código para observar os resultados parciais das regras que foram mantidas.

Em comparação com o paradigma funcional, foi destacado o poder e a generalidade dos predicados da programação lógica, afinal, funções representam apenas um subconjunto das relações no geral. A facilidade obtida com programação de restrições em Prolog permitiu implementar o resolvedor para o quebra-cabeça de maneira mais dinâmica e em muito menos tempo do que em Haskell ou em Lisp. Destaca-se entretanto, que a natureza do problema o torna especialmente compatível com o paradigma utilizado: outros programas eventualmente não compartilhariam de tantas vantagens como este o fez.