Relatório de Implementação do jogo Kakuro (em Prolog)

Alek Frohlich & Nicolas Goeldner, 02 de Dezembro de 2019,

Executando o programa:

Detalhes para como configurar os casos de teste e como executar a aplicação se encontram no README do repositório.

Regras do jogo:

Cada tabuleiro de Kakuro consiste em uma grade em branco com dicas de soma em vários lugares. O objetivo é preencher todos os quadrados vazios usando os números de 1 a 9, para que a soma de cada bloco horizontal seja igual à pista à esquerda e a soma de cada bloco vertical seja igual à pista no topo. Além disso, nenhum número pode ser usado no mesmo bloco mais de uma vez. (tradução literal de: https://www.conceptispuzzles.com/index.aspx?uri=puzzle/kakuro/rules).

Relatório de implementação:

A implementação proposta busca solucionar um tabuleiro de Kakuro, ou em outras palavras, busca uma configuração no qual o tabuleiro não esteja inconsistente. Para isso, definimos uma série de restrições:

1. Os possíveis valores de uma casa branca estão entre 1 e 9.
2. Não podem haver elementos repetidos entre duas casas pretas na mesma coluna ou linha.
3. Uma linha ou uma coluna completa deve ter a soma de seus elementos igual a dica contida na casa preta que a guia.

Casas pretas guiam as casas brancas abaixo e/ou a sua direita. Podemos ver na Figura 1 abaixo que as casas brancas de índice (1,1) e (1,2), as quais contém os elementos 9 e 8, respectivamente, têm seus valores ditados pelas casas pretas de índices (0,1), (0,2) e (1,0), no sentido que 9 e 8 tem que somar 17, 9 e 3 tem que somar 12 e 8,2,1,3 e 7 tem que somar 21.

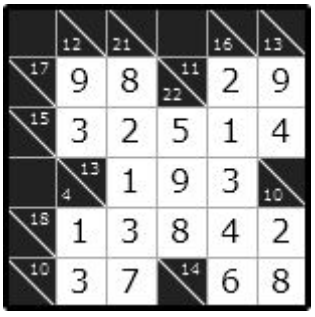

Figura 1. Um tabuleiro de Kakuro.

Para implementar dadas restrições desenvolvemos o predicado *kakuro*(*Rows*), que recebe o tabuleiro de kakuro como uma lista de linhas. Cada linha é uma lista de casas. Cada entrada é, uma lista de três

posições: *Color, First* e *Second. Color* representa a cor da casa (0 para preto e 1 para branco). *First* representa o valor de casas brancas e a dica vertical de casas pretas. *Second* só possui semântica para casas pretas, possuindo o valor de suas dicas horizontais. *kakuro*(*Rows*) pode ser visto na Figura 2.

```
% solves kakuro board
kakuro(Rows) :-
    % make sure white cells are in [1,9]
    maplist(checkWhiteLine, Rows),
    % get columns
    transpose(Rows, Columns),
    % apply constraints over rows and columns
    maplist(checkRawLineSequence, Rows),
    maplist(checkRawColSequence, Columns).
```

Figura 2. Implementação do predicado *kakuro.*

Primeiro, restringimos as casas brancas do tabuleiro para que possuam *First* entre 1 e 9 (Figura 3). *maplist* (de *library(apply))* mapeia um predicado sobre todos elementos de uma lista, se algum resultar em falso, *maplist* também resultará falso. Mapeamos o predicado checkWhite, que utiliza as restrções *in 1..9* e #= sobre cada casa de dada linha. Com o predicado *checkWhiteLine* terminamos de aplicar a primeira restrição sobre o conjunto solução.

```
% constraint white cells to [1,9]
checkWhiteLine(Line) :-
    maplist(checkWhite, Line).
checkWhite([Color,First,_]) :-
    (Color #= 0, !);
    First in 1..9.
%
```

Figura 3. Implementação do predicado *checkWhiteLine.*

Aplicamos o resto das quatro restrições com *checkRawLineSequence* e *checkRawColSequence*:

```
% constraint sum and repetition of sequence.
checkRawLineSequence(RawSequence) :-
    getLineSequence(RawSequence, Sequences),
    maplist(checkSequence, Sequences).

checkRawColSequence(RawSequence) :-
    getColSequence(RawSequence, Sequences),
    maplist(checkSequence, Sequences).

checkSequence([_]) :- !.
checkSequence([Black|Whites]) :-
```

```
    all_different(Whites),
    sum(Whites, #=, Black).
```

Figura 4. Implementação de checkRawLineSequence e checkRawLineSequence.

Similar ao predicado *checkWhiteLine,* os dois *checks* funcionam através do uso de *maplist*. Ambos começam resgatando a lista de sequências brancas guiadas por casa preta contida na respectiva linha/coluna. Os detalhes tal lista (*Sequences)* são ilustrados na Figura 5, por enquanto basta saber o formato dela: [[Black1|Whites1], [Black2], [Black3, White3], ...]. Cada elemento de *Sequences* é uma lista em que o primeiro elemento é preto e os seguintes são brancos. Essa lista é construída da ordem esquerda-direita sobre cada linha/coluna do tabuleiro kakuro. A segunda linha do tabuleiro ilustrado na Figura 1 seria codificada como *Sequences* = [[17,9,8],[11,2,9]]. Sobre cada sequência obtida, aplicamos o predicado *checkSequence. checkSequence* resulta em true caso todos os elementos brancos sejam diferentes, e a soma dos mesmos iguale a dica preta. Assim terminamos de aplicar as restrições.

```
%
% retrieves guided white sequence as a list of list:
% Sequences = [[Black, w1, ..., wn], [Black2],
%       [Black3, w31, ..., w3n], ..., [Blackn, wn1, ..., wnn]]
ehLineGetSequence([],[],[]) :- !.
ehLineGetSequence(BigList, SmallList, [[Color, First, Second]|T3]) :-
    ehLineGetSequence(NewBigList, NewSmallList, T3),
    (
        (Color #= 0, BigList = [[Second|NewSmallList]|NewBigList],
SmallList = [], !) ;
        (SmallList = [First|NewSmallList], BigList = NewBigList)
    ).

ehColGetSequence([],[],[]) :- !.
ehColGetSequence(BigList, SmallList, [[Color, First, _]|T3]) :-
    ehColGetSequence(NewBigList, NewSmallList, T3),
    (
        (Color #= 0, BigList = [[First|NewSmallList]|NewBigList],
SmallList = [], !);
        (SmallList = [First|NewSmallList], BigList = NewBigList)
    ).
%

%
% retrieve guided white sequences
getLineSequence(RawSequence, Sequences) :-
    ehLineGetSequence(Sequences, [], RawSequence).

getColSequence(RawSequence, Sequences) :-
```

```
    ehColGetSequence(Sequences, [], RawSequence).
%
```

Figura 5. Implementação de *getLineSequence* e *getColSequence*.

Programação de restrições:

Neste trabalho fizemos uso de programação por restrições através da biblioteca clp(FD) (Constraint Logic Programming Over Finite Domains). Programação por restrições sobre domínio finito é um paradigma de programação que envolve modelar um problema sobre valores do domínio e então buscar soluções compatíveis com as restrições. Com clp(FD) trabalhamos em domínios inteiros (Z) para os possíveis valores de *First* nas casas brancas. Os predicados que utilizamos para restringir foram: *sum* (Figura 4), *all_different* (Figura 4), *in* (Figura 3) e #= (Figura 3).

Comparação entre paradigmas funcional e lógico:

O paradigma lógico possui a vantagem de que não precisamos implementar toda a parte do backtracking, porém no paradigma funcional as operações com listas foram muito mais fáceis de serem implementadas. Por exemplo, nossa maior dificuldade no trabalho 3 foram os predicados *ehLineGetSequence* e *ehColGetSequence* que apenas juntam em listas alguns elementos de uma lista. No paradigma funcional, a função desse predicado é implementada de uma forma mais fácil e implementar o backtracking no paradigma funcional não foi tão difícil quanto fazer esses predicados.