Universidade Federal de Santa Catarina

Campus de Florianopolis

Relatório Vergleichssudoku - resolução em PROLOG

João Volpato

Thiago Bewiahn

Rafael Vargas

Florianópolis/SC

2022

Relatório Vergleichssudoku Prof. Maicon Zatelli

Resumo

O problema escolhido para ser resolvido neste trabalho foi o Vergleichssudoku. Ele é semelhante ao sudoku tradicional, com as mesmas restrições de não repetir os números em linhas, colunas e quadrados. Pórem a entrada do problema escolhido não contém números mas sim comparadores entre as células do tabuleiro. Com base nessas comparações (maior que e menor que), os números devem ser preenchidos no tabuleiro.

Sumário

1 Solução

1.1 Introdução

Para visualizar a solução so problema é necessário executar o arquivo suduko9x9.pl por meio do comando swipl -f sudoku9x9.pl, depois de abrir o repl deve se executar o comando: solve(Sudoku), append(Sudoku, Vs), labeling([ff], Vs).

Inicialmente a ideia era "traduzir"a solução em Haskell e LISP para Prolog, porém percebemos que não seria viável muito menos ideal, pois não estariámos utilizando o potencial para resolver problemas lógicos deste paradigma.

1.2 Estruturas

O tabuleiro do sudoku foi reprsentado como uma matriz (lista de listas) Solution, e cada célula do tabuleiro possui um váriavel associada. Letras indicam as linhas, e números as colunas.

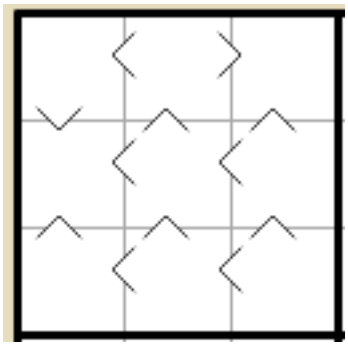
```
Solution = [
[A1, A2, A3, A4, A5, A6, A7, A8, A9],
[B1, B2, B3, B4, B5, B6, B7, B8, B9],
[C1, C2, C3, C4, C5, C6, C7, C8, C9],
[D1, D2, D3, D4, D5, D6, D7, D8, D9],
[E1, E2, E3, E4, E5, E6, E7, E8, E9],
[F1, F2, F3, F4, F5, F6, F7, F8, F9],
[G1, G2, G3, G4, G5, G6, G7, G8, G9],
[H1, H2, H3, H4, H5, H6, H7, H8, H9],
[I1, I2, I3, I4, I5, I6, I7, I8, I9]],
```

Os comparadores foram adicionados a solução não como uma entrada mas sim como parte da lógica da solução.

Por exemplo, essas são as comparações presentes entre as células do primeiro bloco:

O código utilizado para as representar foi:

```
smallerBigger(A1, A2, A3), % line 1
smallerSmaller(B1, B2, B3), % line 2
smallerSmaller(C1, C2, C3), % line 3
biggerSmaller(A1, B1, C1), % column 1
```



```
smallerSmaller(A2, B2, C2), % column 2
smallerSmaller(A3, B3, C3), % column 3
```

smallerBigger de A1, A2, A3 impõem a restrição que A1 < A2 e A2 > A3;

smallerSmaller de B1, B2, B3 impõem a restrição que B1 < B2 e B2 < B3;

biggerSmaller de C1, C2, C3 impõem a restrição que C1 > C2 e C2 < C3.

Essa lógica se repete para cada bloco presente no problema selecionado, que está em: <https://www.janko.at/Raetsel/Sudoku/Vergleich/011.a.htm>

1.3 Comparação: Haskell X LISP X Prolog

Implementação na linguagem Prolog foi mais facíl do que utilizando Haskell e Lisp, por conta do paradigma lógico e também pelo fato de utilizarmos métodos prontos da biblioteca clpfd. Além disso foi a solução mais performática. A forma de imaginar a solução do problema foi completamente diferente também, e acabamos optando por uma representação diferente dos outros trabalhos. Isso gerou problemas, como citado na seção Dificuldades.

1.4 Algoritmos

Para achar a solução restringimos os valores que cada célula pode assumir, depois criamos os blocos, restringimos que linhas, colunas e blocos não podem ter valores repetidos (restrições do sudoku original). Depois restringimos os valores de acordo com a lógica dos comparadores explicada previamente.

O código abaixo mostra um trexo do programa exemplificando as restrições do sudoku:

```
flatten(Solution, Tmp), Tmp ins 1..9, % All elements are between 1 and 9
Rows = Solution,
transpose(Rows, Columns),
```

```
blocks(Rows, Blocks),
maplist(all_different, Rows),
maplist(all_different, Columns),
maplist(all_different, Blocks),

% Solution is presented in blocks (like the ones in sudoku), starting at the
    top left corner
% Block 1
smallerBigger(A1, A2, A3), % line 1
smallerSmaller(B1, B2, B3), % line 2
smallerSmaller(C1, C2, C3), % line 3
biggerSmaller(A1, B1, C1), % column 1
smallerSmaller(A2, B2, C2), % column 2
smallerSmaller(A3, B3, C3), % column 3
```

Dentro de cada predicado de comparação, é feita uma divisão da comparação em smaller ou bigger, chamando esses predicados e dentro deles é feita a verificação de validade dos números com o predicado valid.

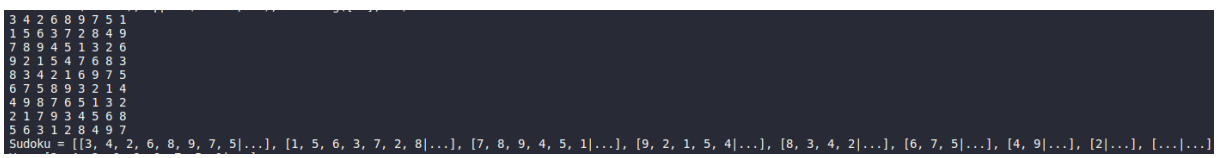
```
valid(A) :-
    member(A, [1, 2, 3, 4, 5, 6, 7, 8, 9]).

bigger(A, B) :-
  valid(A),
  valid(B),
  A > B.

smaller(A, B) :-
  valid(A),
  valid(B),
  A < B.
```

1.5 Saída

O programa imprime a matriz resposta ao encontrá-la, e a variável passada ao predicado solve recebe o valor dessa matriz, conforme o exemplo abaixo:



1.6 Organização

A comunicação do grupo foi feita com "reuniões"no tempo de aula e em um grupo de Whatsapp.

1.7 Dificuldades

A principal dificuldade foi como representar os comparadores no código. Inicialmente tentamos manter o mesmo formato utilizado em Haskell e LISP, porém devido à mudança de paradigma isso se provou muito complicado. Strings e chars pareciam não se comportar da mesma forma. Então a solução foi explicitar as comparações dentro do predicado de solução.

Após isso, tivemos problemas com performance. Isso foi resolvido com 2 alterações: inicialmente o predicado maplist utilizava all_distinct, que utiliza um processamento mais pesado para eliminar as possibilidades dos conjuntos. Ele acabou sendo substituído por all_different, que remove menos possibilidades de uma vez porém é suficiente para o caso do sudoku. Isso em conjunção com a mudança do labeling (estratégia de preenchimento) para first-fail melhorou suficientemente a performance e nos permitiu executar o programa até encontrar uma solução.