UNIVERSIDADE FEDERAL DE SANTA CATARINA

Paradigmas de Programação - INE5416

Professor: Maicon Rafael Zatelli

Alunos:

Allan Soares Silva	(19200410)
Antonio Silverio Montagner	(19203742)
Eduardo Vinicius Betim	(19203161)

Trabalho III - Programação Lógica - Prolog

Puzzle Vergleichssudoku

Florianópolis

Dezembro de 2022

1. PROBLEMA ESCOLHIDO

O problema abordado neste trabalho foi o puzzle Vergleichssudoku. O puzzle possui sinais de relação de maior e menor entre duas células, considerando que para a célula qual a seta aponta é possui um número menor que o na outra célula, assim estabelecendo uma relação entre elas. Os símbolos de relação entre células são “<“, “>”, “^”, “v”.

Cada uma das células do puzzle possuem de duas a quatro setas nas divisas entre si, assim representando a comparação que deve ser feita.

O puzzle tratado possui o tamanho de 9x9, que é 9 linhas por 9 colunas, possuindo no total 6 regiões de dimensões 3x3. Escreva números de 1 a N nas células da grade de tamanho NxN, de modo que cada número ocorre exatamente uma vez em cada linha, em cada coluna e em cada região.

Além de respeitar a comparação apresentada, um número não pode se repetir numa mesma linha, coluna ou bloco.

2. SOLUÇÃO ESCOLHIDA

Para resolver o problema apresentado, o algoritmo trata o tabuleiro como uma lista de 81 células e percorre ela célula a célula, validando os números presentes na célula acima e à esquerda da célula atual. Também existe uma matriz 9x9 das possibilidades de cada célula. Quando o algoritmo chega em uma célula, ele calcula as possibilidades de números que podem preenchê-la e guarda-as em uma lista na matriz de possibilidades, no índice equivalente a célula atual.

A ideia dessa solução é baseada em um algoritmo regular para solucionar um tabuleiro de sudoku, mas também validando as células adjacentes enquanto o tabuleiro é explorado, seguindo a regra de maior/menor presente na variação Vergleich. Para isso, o grupo se reuniu em uma plataforma online e, em conjunto, desenvolveu a solução apresentada.

2.1 ENTRADA DE DADOS

Ao executar o programa, é pedido o '*Board id:*' do qual é o identificador do tabuleiro (sem sua extensão '.txt') do qual será solucionado. Para isso, é necessário ter o arquivo do tabuleiro devidamente salvo na pasta 'boards'.

A entrada é por meio da leitura e *parsing* de um arquivo de texto no formato da Figura 1, que representa o tabuleiro de um puzzle solucionável. O arquivo contém 9 linhas, cada uma com uma sequência de 9 pares de caracteres que representam as operações à direita e abaixo, respectivamente, de cada célula. Uma célula com a dupla **+\-**, por exemplo, indica que a mesma deve possuir um valor maior que a célula à direita e menor que a célula abaixo.

Existe também o identificador **.** que indica que a célula não possui uma condição em relação à célula da direita, caso seja o primeiro caractere da dupla, ou em relação à célula abaixo, caso seja o segundo caractere da dupla.

Por fim, será retornada uma matriz, parecida com o da Figura 3, com a solução do puzzle passado. A solução é retornada diretamente na interface de execução.

```
++ ++ .- +- ++ .- ++ +- .-
-- -+ .- ++ -+ .+ ++ +- .-
+. -. .. +. -. .. -. -. ..
++ -- .- +- -- .+ ++ +- .+
+- -- .- -+ ++ .- -+ ++ .-
-. +. .. -. +. .. +. -. ..
-- -+ .+ -- +- .- ++ -- .+
-- +- .- -- ++ .+ -+ ++ .+
-. +. .. +. +. .. -. -. ..
```

Figura 1: Formato dos dados de entrada, puzzle 111.

```
- -  - -  .-  -+  ++  .-  ++  ++  .-
- -  ++  .-  -+  - -  .+  - -  -+  .-
+.  -.  ..  -.  -.  ..  -.  -.  ..
- -  - -  .-  ++  ++  .+  - -  +-  .-
- -  ++  .-  ++  +-  .-  ++  ++  .-
+.  -.  ..  -.  +.  ..  +.  -.  ..
- -  +-  .+  -+  ++  .-  +-  - -  .+
+-  +-  .+  +-  - -  .-  -+  ++  .+
+.  +.  ..  +.  -.  ..  -.  +.  ..
```

Figura 2: Formato dos dados de entrada, puzzle 156.

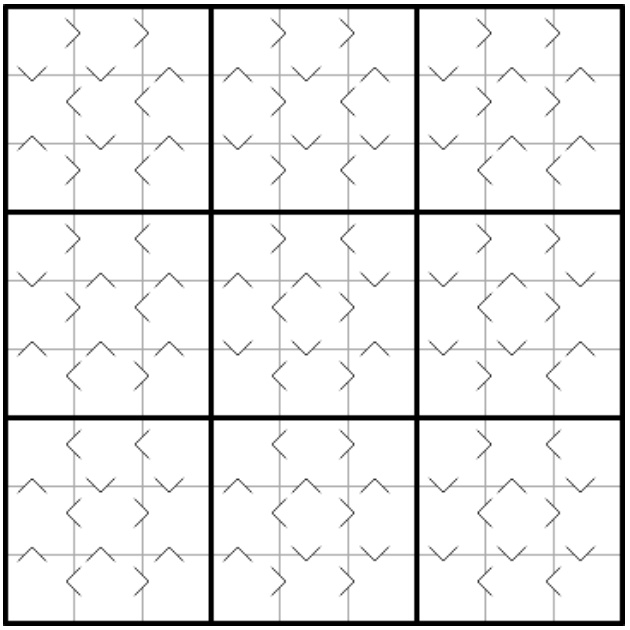

Figura 3: Imagem do puzzle número 111.

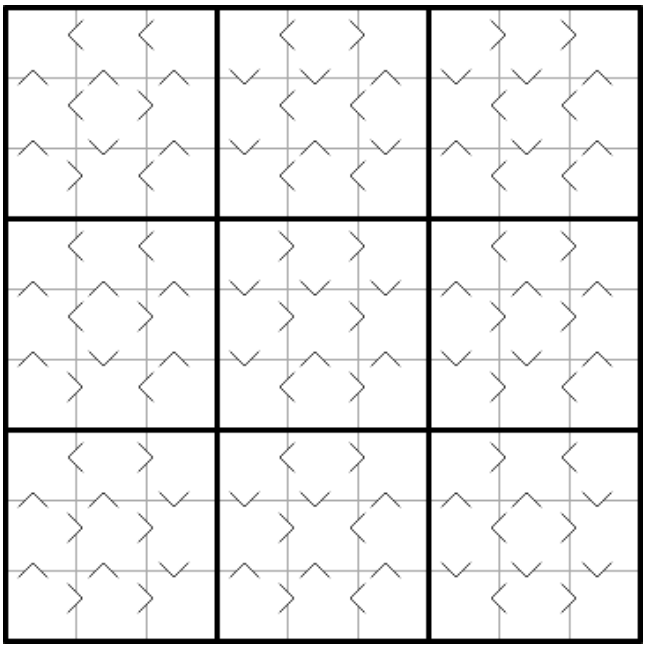

Figura 4: Imagem do puzzle número 156.

```
9++ 7++ 5.-   6+- 4++ 3.-   8++ 2+- 1.-
1-- 4-+ 6.-   8++ 2-+ 9.+   7++ 5+- 3.-
3+. 2-. 8..   5+. 1-. 7..   4-. 6-. 9..

6++ 1-- 2.-   4+- 3-- 8.+   9++ 7+- 5.+
5+- 3-- 4.-   7-+ 9++ 1.-   6-+ 8++ 2.-
8-. 9+. 7..   2-. 6+. 5..   3+. 1-. 4..

2-- 6-+ 9.+   1-- 7+- 4.-   5++ 3-- 8.+
4-- 5+- 1.-   3-- 8++ 6.+   2-+ 9++ 7.+
7-. 8+. 3..   9+. 5+. 2..   1-. 4-. 6..
```

Figura 5: Imagem da solução para o puzzle número 111, feito em Lisp, retornado ao usuário.

```
1-- 2-- 3.-   6-+ 7++ 4.-   9++ 8++ 5.-
5-- 8++ 7.-   2-+ 3-- 9.+   1-- 4-+ 6.-
6+. 4-. 9..   1-. 5-. 8..   2-. 3-. 7..

2-- 3-- 5.-   9++ 8++ 7.+   4-- 6+- 1.-
4-- 9++ 6.-   5++ 2+- 1.-   8++ 7++ 3.-
7+. 1-. 8..   4-. 6+. 3..   5+. 2-. 9..

3-- 5+- 4.+   7-+ 9++ 2.-   6+- 1-- 8.+
8+- 6+- 2.+   3+- 1-- 5.-   7-+ 9++ 4.+
9+. 7+. 1..   8+. 4-. 6..   3-. 5+. 2..
```

Figura 6: Imagem da solução para o puzzle número 156, feito em Lisp, retornado ao usuário.

2.2 ELABORAÇÃO

2.2.1 Estruturas e tipos de dados utilizadas

O tipo de dado Board representa uma célula e possui os campos value, right e bottom que representam o valor, o char correspondente a operação na parte de baixo da célula e o char correspondente a operação do lado direito da célula.

```
board)[
    ['++', '++', '.-', '+-', '++', '.-', '++', '+-', '.-'],
    ['--', '-+', '.-', '++', '-+', '.+', '++', '+-', '.-'],
    ['+.', '-.', '..', '+.', '-.', '..', '-.', '-.', '..'],
    ['++', '--', '.-', '+-', '--', '.+', '++', '+-', '.+'],
    ['+-', '--', '.-', '-+', '++', '.-', '-+', '++', '.-'],
    ['-.', '+.', '..', '-.', '+.', '..', '+.', '-.', '..'],
    ['--', '-+', '.+', '--', '+-', '.-', '++', '--', '.+'],
    ['--', '+-', '.-', '--', '++', '.+', '-+', '++', '.+'],
    ['-.', '+.', '..', '+.', '+.', '..', '-.', '-.', '..']
].
```

2.2.2 Manipulação dos dados

Ao longo do código o tabuleiro é tratado às vezes como uma matriz de ordem 9, e às vezes como uma lista de 81 elementos. Isso é feito apenas por conveniência, já que algumas operações são mais fáceis de serem implementadas como lista. As funções a seguir são responsáveis pela conversão entre uma estrutura e outra:

```
% Transforma uma matriz em lista
list_to_board([], _, []).
list_to_board(Board, Size, [List|Rest]):-
    list_to_board_aux(Board, Size, List, Tail),
    list_to_board(Tail, Size, Rest).

list_to_board_aux(Tail, 0, [], Tail).
list_to_board_aux([Item|Board], Size, [Item|List],
Tail):-
    NSize is Size - 1,
    list_to_board_aux(Board, NSize, List, Tail).

% Transforma uma lista em uma matriz
board_to_list([], _, []).
board_to_list(List, T, [Start|Rest]) :-
    append(Start, Remainder, List),
    length(Start, T),
    board_to_list(Remainder, T, Rest).
```

O *fallback* de *partialOp* é uma função parcial que, independente do segundo valor aplicado, sempre retornará verdadeiro. Ele será utilizado em casos onde a célula não possui uma operação em algum dos lados.

As outras funções principais utilizadas na manipulação de dados são responsáveis por retornar as células presentes na linha, coluna e região de uma dada coordenada.

```
% Retorna a linha de um tabuleiro em um dado índice.
row_at(Board, I, Row) :- nth0(I, Board, Row).

% Retorna a coluna do tabuleiro em um dado índice
transpondo-o e utilizando a
% `row_at`
column_at(Board, I, Column) :-
    transpose(Board, T),
```

```
        row_at(T, I, Column).

    % Retorna a região do tabuleiro a qual um dado ponto
(x, y) pertence
    region_at(Board, [X|Y], Region) :-
        Y_ is Y // 3 * 3,
        X_ is X // 3 * 3,
        drop(X_, Board, B),
        take(3, B, Rows),
        board_to_list(Region, 3, Rows).
```

2.2.3 Elaboração do solucionador do problema

Percorre o tabuleiro e determina os números que devem preencher cada célula, usando a função possibilities para determinar os valores possíveis em cada iteração.

As possibilidades são calculadas gerando uma lista no intervalo [1, 9] e removendo os valores já utilizados na linha, coluna e região em que a célula está presente. Após isso, a lista é filtrada novamente com base na aplicação da operação *bottom* da célula acima da atual e na operação *right* da célula à esquerda da atual utilizando o valor da célula atual em ambas.

```
possibilities(Board, I, List) :-
    itop(I, Coord),
    list_to_board(Board, 9, B),
    cell_at(B, Coord, CurrentCell),
    (Y > 0 -> cell_at(B, [X, Y - 1], UpperCell);
create_empty_cell(UpperCell)),
    (X > 0 -> cell_at(B, [X - 1, Y], LeftCell);
create_empty_cell(LeftCell)),
    row_at(Board, I, Row),
    column_at(Board, I, Column),
    region_at(Board, I, Region),
    Shared = [Row, Column, Region],
    maplist(value, Shared, UsedValues),
```

```
        right_op(LeftCell, RightOp),
        inverse_of(RightOp, InvRightOpLeftCell),
        bottomt_op(UpperCell, BottomtOp),
        inverse_of(BottomtOp, InvBottomtOpUpperCell),
        right_op(CurrentCell, RightOpCurr),
        bottom_op(CurrentCell, BottomOpCurr),
        Operations          =          [InvRightOpLeftCell,
InvBottomtOpUpperCell, RightOpCurr, BottomOpCurr],
        count('+', Operations, GreaterQuantity),
        count('-', Operations, LessQuantity),
        MinValue is 1 + GreaterQuantity,
        MaxValue is 9 - LessQuantity,

        not_a_member(X, Possibilities), X >= MinValue, X <=
MaxValue,
        maplist(Possibilities, List).
```

Os passos para solucionar são:

0. O índice inicia em 0 e a matriz de possibilidades inicia vazia
1. Verifica o valor do índice
    1.1 O valor é -1 ou 81
    1.1.1 Passo 4
2. O algoritmo está incrementando ou decrementando?
    2.1 Incrementando
    2.1.1 Calcula as possibilidades da célula atual
    2.1.2 Passo 3
    2.2 Decrementando
    2.2.1 Busca as possibilidades restantes disponível na matriz para a célula atual
3. Verifica as possibilidades
    3.1. Há possibilidades

3.1.1. Substitui o valor da célula no tabuleiro pela primeira possibilidade calculada

3.1.2. Guarda o resto das possibilidades na matriz de possibilidades

3.1.3. Vai para a próxima célula

3.1.4. Passo 1

3.2. Não há possibilidades

3.2.1 Zera o valor da célula atual no tabuleiro

3.2.2 Retorna à célula anterior

3.2.3 Passo 1

4. Fim do algoritmo.

```
solve(Board, Solved) :-
     board_to_list(Board, 9, B),
     possibilities(B, PossMatrix)
     go(B, PossMatrix, 0, yes, Solved).

go(Board, PossMatrix, I, Forward, Result) :- !.
     (I = -1; I = 81 -> list_to_board(Board, Solved),
Result = Solved
     ;
     (Forward = yes -> possibilities(Board, I, Poss);
nth0(I, PossMatrix, Poss))
     ),
     length(Poss, PossQuantity),
     (PossQuantity > 0 ->
     nth0(I, Board, CurrCell),
     create_empty_cell(EmptyCell),
     replace(Board, EmptyCell, I, ReplacedBoard),
     go(ReplacedBoard, PossMatrix, I - 1, no)
     ;
     take(1, Poss, X),
     drop(1, Poss, XS),
     nth0(I, Board, CurrCell),
     right(CurrCell, RightOp),
     bottom(CurrCell, BottomOp),
     create_new_cell(X, RightOp, BottomOp, NewCell),
     replace(Board, NewCell, I, ReplacedBoard),
     replace(PossMatrix, XS, I, ReplacedPossMatrix),
     go(ReplacedBoard, ReplacedPossMatrix, I + 1, yes)
     ).
```

3. PROBLEMAS ENCONTRADOS

Dificuldades ao aprender Prolog por ser um paradigma completamente diferente do imperativo, orientado a objetos ou funcional. A falta de conhecimento acabou fazendo com que não conseguíssemos implementar o solucionador de forma correta.