Universidade Federal de Santa Catarina
Departamento de Informática e Estatística (INE)
Curso: Ciência da Computação
Disciplina: Paradigmas de Programação (INE5416)
Professor: Maicon Rafael Zatelli
Estudante: Guilherme Adenilson de Jesus (22100620)

Relatório T3 - Prolog

- **Problema**

O presente relatório é referente ao desenvolvimento de um programa em Prolog para um solucionador do quebra-cabeça escolhido, Vergleichssudoku (Sudoku Greater Than). Esse *puzzle* é uma variante do Sudoku, em que cada posição do tabuleiro possui comparadores para as posições adjacentes a ela dentro do seu quadrado.

- **Solução**

O tabuleiro é separado em três matrizes. Uma para armazenar os valores de cada posição, uma para os comparadores horizontais e outra para os verticais. Essas estruturas utilizam o List padrão como auxílio para armazenar os valores.

As três matrizes são enviadas para o predicado *solve_sudoku*. Que testará cada alternativa possível. Em ordem do exposto na Figura 1, ele possui 4 cláusulas:

1. Quando o tabuleiro estiver concluído, imprime o resultado;
2. Ao chegar no fim da linha (Col = 9), atualiza o valor de Row e volta Col para o 0, chamando o predicado para os novos valores;
3. Se a posição estiver ocupada, faz Col + 1 e ativa o predicado novamente;
4. Quando a posição está vazia (= 0), gera uma sequência de 1 a 9 e avalia se algum desses valores é válido pelo is_valid. Se sim, substitui o tabuleiro e vai à próxima posição. Se não, ou em algum ponto da recursão der errado, avalie a próxima sugestão.

```
% Se chegar no fim do Board, printa a solução
solve_sudoku(Board, 9, _, _, _) :- print_sudoku(Board), !.

% Chegou no fim de uma linha, vai para a próxima
solve_sudoku(Board, Row, 9, CompH, CompV) :-
    NewRow is Row + 1,
    solve_sudoku(Board, NewRow, 0, CompH, CompV).

% Se a posição atual já está preenchida, vai para a próxima posição
solve_sudoku(Board, Row, Col, CompH, CompV) :-
    nth0(Row, Board, R), nth0(Col, R, Num), Num \= 0,
    NewCol is Col + 1,
    solve_sudoku(Board, Row, NewCol, CompH, CompV).

solve_sudoku(Board, Row, Col, CompH, CompV) :-
    % Se a posição atual está vazia, busca um palpite válido
    nth0(Row, Board, R), nth0(Col, R, Num), Num = 0,
    between(1, 9, Guess),
    is_valid(Board, Row, Col, Guess, CompH, CompV),

    % Substitua o 0 pelo palpite válido
    replace_in_row(R, Col, Guess, NewR), replace_in_row(Board, Row, NewR, NewBoard),

    % Vai para próxima posição
    NewCol is Col + 1,
    solve_sudoku(NewBoard, Row, NewCol, CompH, CompV).
```

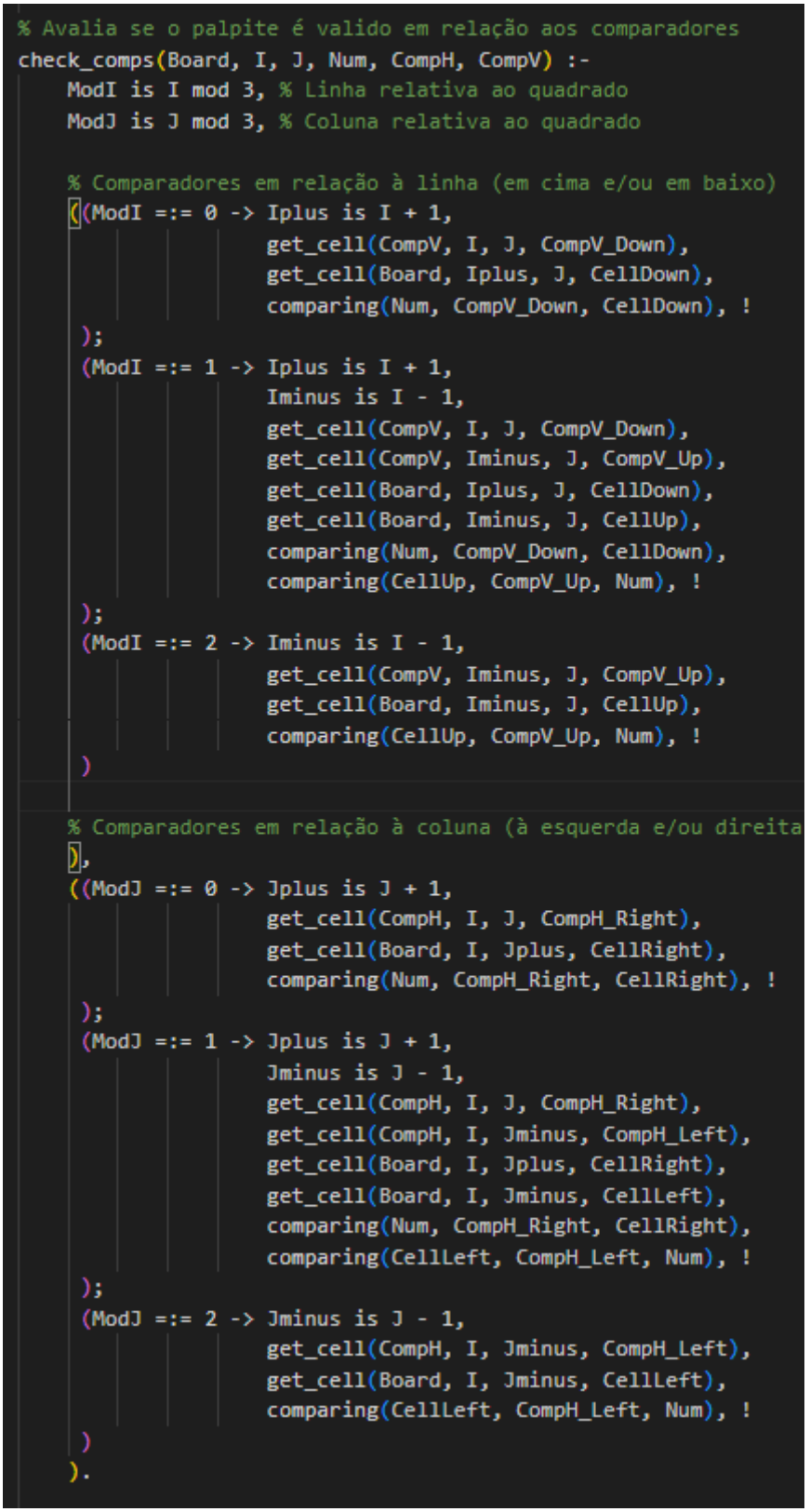
Figura 1 - Predicado *solve_sudoku*
Fonte: autoria própria

A seguir, há o predicado *is_valid*. Ela avaliará se a alternativa solicitada (Num) será possível baseada na atual organização do tabuleiro. Analisando o trecho de código abaixo (Figura 2), são feitas até 4 verificações: se Num já é um elemento da linha, coluna e quadrado. Essas análises são iguais às realizadas no Sudoku, o que altera é a última, que utiliza os comparadores referentes à posição pelo predicado *check_comps*.

```
% Avalia se o palpite é válido
is_valid(Board, I, J, Num, CompH, CompV) :- nth0(I, Board, Row),
                                            (
                                            (is_elem(Row, Num), !, false); % Palpite já está na linha?
                                            (get_square(Board, I, J, Square), is_elem(Square, Num), !, false);
                                            (get_col(J, Board, Col), is_elem(Col, Num), !, false); % Palpite já
                                            (check_comps(Board, I, J, Num, CompH, CompV), !) % O palpite é váli
                                            ).
```

Figura 2 - Precidado *is_valid*
Fonte: autoria própria

A partir disso, há o predicado *check_comps*, que como dito anteriormente, avalia os comparadores da posição referente. Destaca-se que cada célula de um quadrado tem diferentes quantidades de comparadores, o que torna necessário a avaliação da posição em relação ao seu quadrado. Para realizar essas verificações, é utilizado o predicado *comparing* para cada comparador.



```
% Avalia se o palpite é valido em relação aos comparadores
check_comps(Board, I, J, Num, CompH, CompV) :-
    ModI is I mod 3, % Linha relativa ao quadrado
    ModJ is J mod 3, % Coluna relativa ao quadrado

    % Comparadores em relação à linha (em cima e/ou em baixo)
    ((ModI =:= 0 -> Iplus is I + 1,
                    get_cell(CompV, I, J, CompV_Down),
                    get_cell(Board, Iplus, J, CellDown),
                    comparing(Num, CompV_Down, CellDown), !
     );
     (ModI =:= 1 -> Iplus is I + 1,
                    Iminus is I - 1,
                    get_cell(CompV, I, J, CompV_Down),
                    get_cell(CompV, Iminus, J, CompV_Up),
                    get_cell(Board, Iplus, J, CellDown),
                    get_cell(Board, Iminus, J, CellUp),
                    comparing(Num, CompV_Down, CellDown),
                    comparing(CellUp, CompV_Up, Num), !
     );
     (ModI =:= 2 -> Iminus is I - 1,
                    get_cell(CompV, Iminus, J, CompV_Up),
                    get_cell(Board, Iminus, J, CellUp),
                    comparing(CellUp, CompV_Up, Num), !
     )

    % Comparadores em relação à coluna (à esquerda e/ou direita
    ),
    ((ModJ =:= 0 -> Jplus is J + 1,
                    get_cell(CompH, I, J, CompH_Right),
                    get_cell(Board, I, Jplus, CellRight),
                    comparing(Num, CompH_Right, CellRight), !
     );
     (ModJ =:= 1 -> Jplus is J + 1,
                    Jminus is J - 1,
                    get_cell(CompH, I, J, CompH_Right),
                    get_cell(CompH, I, Jminus, CompH_Left),
                    get_cell(Board, I, Jplus, CellRight),
                    get_cell(Board, I, Jminus, CellLeft),
                    comparing(Num, CompH_Right, CellRight),
                    comparing(CellLeft, CompH_Left, Num), !
     );
     (ModJ =:= 2 -> Jminus is J - 1,
                    get_cell(CompH, I, Jminus, CompH_Left),
                    get_cell(Board, I, Jminus, CellLeft),
                    comparing(CellLeft, CompH_Left, Num), !
     )
    ).
```

Figura 3 – Predicado check_comps
Fonte: autoria própria

Como visto na Figura 4 abaixo, o predicado *comparing* recebe dois inteiros e uma string (comparador). Para acelerar as análises, é feito algumas avaliações das alternativas extremas (1 e 9). Se o valor 1 estiver ao lado de um comparador ‘>’, é impossível que haja outro número menor que ele, então já é considerado False de

imediato, mesma coisa para valor 9 com ‘<’. Essa verificação premeditada é importante devido ao valor 0 (posição vazia), que qualquer valor comparado a ele já se tornaria True, podendo considerar uma alternativa que nunca daria uma solução correta. Caso passe por todas essas avaliações, é feito a comparação entre a e b.

```
% Compara se A op B, sendo op = > ou <
comparing(1, '>', _) :- false, !. % 1 > Any é sempre falso
comparing(9, '<', _) :- false, !. % 9 < Any é sempre falso
comparing(_, '<', 1) :- false, !.
comparing(_, '>', 9) :- false, !.
comparing(0, _, _) :- !. % 0 é valor para posição vazia, então sempre verdade
comparing(_, _, 0) :- !.
comparing(A, '>', B) :- A > B, !. % Comparações em si
comparing(A, '<', B) :- A < B, !.
```

Figura 4 - Predicado *comparing*
Fonte: autoria própria

- **Entrada pelo usuário**

Como exposto na Figura 5 abaixo, o usuário enviará pelo SWIPL o arquivo de texto referente à configuração do quebra-cabeça a partir do predicado *solve*.

```
% Recebe um nome de arquivo que contem os comparadores, separa-os em
% horizontais e verticais e inicia a busca pela solução
solve(Filename) :-
    (init_board(Board),
    read_file(Filename, Lines),
    slice(Lines, 0, 9, CompH),
    slice(Lines, 10, 9, CompV),
    solve_sudoku(Board, 0, 0, CompH, CompV), !);
    write('Solucao nao encontrada'), false, !.
```

Figura 5 - Recebimento de entrada
Fonte: autoria própria

O arquivo de configuração é similar ao apresentado na Figura 6 a seguir. Começa pelos comparadores horizontais, linha a linha. Quando não houver um comparador, é utilizado o caracter “|”, indicando troca de quadrado ou fim da linha do quebra-cabeça. Para facilitar a leitura e separação, antes da descrição dos comparadores verticais é colocado uma linha com um único caracter “.”.

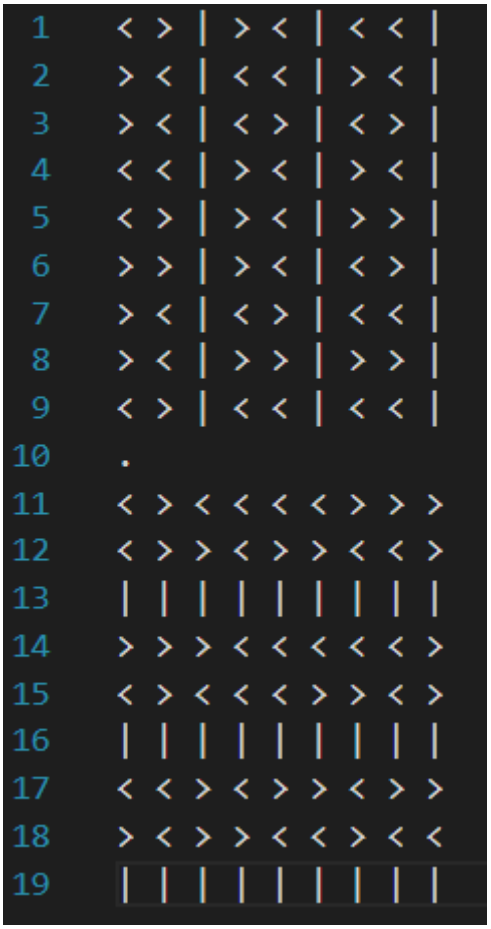


Figura 6 - Configuração do quebra-cabeça
Fonte: autoria própria

O resultado do jogo é imprimido no SWIPL (Figura 7), mostrando os valores que estarão presentes em cada posição do quebra-cabeça. Caso não haja uma solução possível, será imprimido um *"Solucao nao encontrada"*.

```
1 ?- solve('puzzle97.txt').
[1,2,3,4,5,6,7,8,9]
[4,5,6,7,8,9,1,2,3]
[7,8,9,1,2,3,4,5,6]
[2,1,4,3,6,5,8,9,7]
[3,6,5,8,9,7,2,1,4]
[8,9,7,2,1,4,3,6,5]
[5,3,1,6,4,2,9,7,8]
[6,4,2,9,7,8,5,3,1]
[9,7,8,5,3,1,6,4,2]
true.
```

Figura 7 - Resultado do quebra-cabeça
Fonte: autoria própria

- **Dificuldades encontradas**

Uma das dificuldades encontradas foi como fazer a leitura do quebra-cabeça a ser resolvido por um arquivo. Foi necessário o auxílio da biblioteca *readutil* para ler o arquivo recebido. Além disso, foi preciso de 3 predicados: *read_file*, *read_lines* (lê a Stream; o arquivo) e *slice* (para separar os comparadores horizontais dos verticais). Todos eles estão presentes num arquivo separado para facilitar a leitura do programa principal (Figura 8).

```
:- use_module(library(readutil)).

read_file(FileName, Lines) :-
    open(FileName, read, Stream),
    read_lines(Stream, Lines),
    close(Stream).

read_lines(Stream, []) :-
    at_end_of_stream(Stream), !.
read_lines(Stream, [Line|Lines]) :-
    \+ at_end_of_stream(Stream),
    read_line_to_string(Stream, LineString),
    string_chars(LineString, LineWithSpaces),
    exclude(=(' '), LineWithSpaces, Line),
    read_lines(Stream, Lines).

slice(List, Start, Size, Slice) :-
    length(Prefix, Start),
    append(Prefix, Rest, List),
    length(Slice, Size),
    append(Slice, _, Rest).
```

Figura 8 - Leitura do arquivo
Fonte: autoria própria