Bosnian Snake

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação Lógica

Turma 3MIEIC05
Grupo Bosnian Snake_2
David Falcão - up201506571
Pedro Miranda - up201506574

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

23 de Dezembro de 2017

Resumo

Este relatório complementa o segundo projeto da Unidade Curricular de Programação Lógica, do Mestrado Integrado em Engenharia Informática e Computação. O projeto consiste na elaboração de um programa, escrito em Prolog, capaz de resolver o problema Bosnian Snake.

Conteúdo

1 Introdução **3**

2 Descrição do Problema **4**

3 Abordagem **4**
3.1 Variáveis de Decisão 4
3.2 Restrições 5
3.3 Estratégia de Pesquisa 7

4 Visualização da Solução **8**

5 Resultados **9**

6 Conclusão **10**

7 Referências **11**

8 Anexos **12**
8.1 Código Fonte 12
8.1.1 bosnian.pl 12
8.1.2 logic.pl 14
8.1.3 interface.pl 19
8.1.4 generator.pl 21
8.1.5 utils.pl 23

1 Introdução

Este projeto tem como principais objetivos a resolução de problemas com diferentes tamanhos de tabuleiro e diferente número de peças, no nosso caso, o puzzle Bosnian Snake.

Este problema consiste em dados um ponto inicial e um ponto final (aleatórios), ser construído um caminho de ligação entre ambos respeitando restrições impostas, seja no meio da board ou nas colunas/linhas.

O relatório terá a seguinte estrutura:

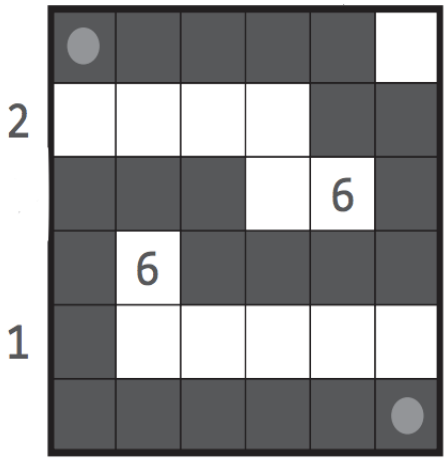
- **Introdução** - Descrição sucinta do projeto e seus objetivos
- **Descrição do Problema** - Descrição detalhada do problema
- **Abordagem** - Descrição da modelação do problema de satisfação de restrições
  - **Variáveis de Decisão** - Descrição das variáveis de decisão
  - **Restrições** - Descrição das variáveis rígida e flexível do problema e a sua implementação.
  - **Estratégia de Pesquisa** - Descrição da estratégia implementada na ordenação de variáveis.
- **Visualização da Solução** - Explicar os predicados que permitem a visualização da solução em mode de texto.
- **Resultados** - Demonstração de exemplos da aplicação em problemas de diferentes complexidades e análise dos resultados obtidos.
- **Conclusão** - Conclusões sobre o projeto.
- **Bibliografia**
- **Anexos**

2 Descrição do Problema

O problema Bosnian Snake consiste em representar uma cobra com 1 célula de largura num tabuleiro sendo que a cabeça e a cauda são posições predefinidas.

A cobra não se pode tocar nunca, nem mesmo diagonalmente. Existem ainda números tanto dentro do tabuleiro como fora.Os números dentro do tabuleiro indicam quantas casas à volta da casa onde o número está vão estar ocupadas por partes da cobra.Os números colocados em linhas ou colunas indicam quantas casas naquela linha/coluna vão ser ocupadas por partes da cobra.

Como podemos ver no exemplo abaixo,há duas células que tem 6 casas ocupadas à sua volta.Há ainda duas linhas com indicação de quantas células ocupadas terá a mesma.



3 Abordagem

3.1 Variáveis de Decisão

A única variável de decisão utilizada é uma lista de tamanho Size*Size, em Size é o tamanho de uma matriz quadrada. Apesar de a variável de decisão ser uma lista, posteriormente é convertida para uma matriz e o problema é todo resolvido tomando-a como matriz. O método de conversão foi o seguinte:

```
1  % Convert a list to matrix
2  % list_to_matrix(+list, +matrix_size, -matrix)
3  list_to_matrix([], _, []).
4
5  list_to_matrix(List, Size, [Row|Matrix]):-
6          list_to_matrix_row(List, Size, Row, Tail),
7          list_to_matrix(Tail, Size, Matrix).
8
9  list_to_matrix_row(Tail, 0, [], Tail).
10
11 list_to_matrix_row([Item|List], Size, [Item|Row], Tail):-
12         NSize is Size-1,
13         list_to_matrix_row(List, NSize, Row, Tail).
```

3.2 Restrições

Em primeiro lugar, na inicialização da variável de decisão foi imposto do domínio [0,1], em que 1 representa os locais por onde passa e 0 os locais por onde não passa a cobra.

De seguida restringimos:

- a soma das linhas indicadas no problema:

```
% Test all horizontal restrictions of the board
% testHorizontalSum(+ board)
testHorizontalSum(Board):-
    findall(Line-Sum, sumLine(Line,Sum),All_Res),
    testHorizontalSum(Board, All_Res).

testHorizontalSum(Board, [Line-Sum|T]):-
    getElemsLine(Board, Line, Elems),
    sum(Elems, #=, Sum),
    testHorizontalSum(Board, T).

testHorizontalSum(_, []).
```

- a soma das colunas indicadas no problema:

```
% Test all vertical restrictions of the board
% testHorizontalSum(+ board)
testVerticalSum(Board):-
    findall(Column-Sum, sumCol(Column,Sum),All_Res),
    testVerticalSum(Board, All_Res).

testVerticalSum(Board, [Column-Sum|T]):-
    getElemsColumn(Board, Column, Elems),
    sum(Elems, #=, Sum),
    testVerticalSum(Board, T).

testVerticalSum(_, []).
```

- a soma das posições em redor: - a soma dos elementos em redor de uma casa tem que ser exatamente aquela referida no enunciado do problema

```
% Test all around position restrictions of the board
% testHorizontalSum(+ board)
testNeighboursSum(Board):-
    findall([Line, Column,Sum], sumAround(Line,Column,Sum),All_Res),
    testNeighboursSum(Board, All_Res).

testNeighboursSum(Board, [Head|T]):-
    nth0(0, Head, Line),
    nth0(1, Head, Column),
    nth0(2, Head, Sum),
    nth0(Line, Board, Temp),
    nth0(Column, Temp, Position),
```

```
    Position #= 0,
    getElemsAround(Board, Line, Column, Elems),
    sum(Elems, #=, Sum),
    testNeighboursSum(Board, T).

testNeighboursSum(_, []).
```

- a soma das posições adjacentes, de modo que haja conectividade da cobra: - para que haja conectividade, se uma casa estiver a 1, tem que ter 2 casas adjacentes a 1 também.

```
% Teste the board connectivity
% All pieces of the snake must have 2 adjacente pieces, except the borders
% that must have only 1
% testBoardConnection(+ board)
testBoardConnection(Board):-
    findall(Line-Column, (nth0(Line, Board, Tmp),nth0(Column, Tmp, _)), Indexes),
    testBoardConnection(Board, Indexes).

testBoardConnection(Board, [Line-Column|T]):-
    checkBorders(Line, Column),
    getElemsAdjacent(Board, Line, Column, Elems),
    sum(Elems, #=, 1), % only has 1 connection
    testBoardConnection(Board, T).

testBoardConnection(Board, [Line-Column|T]):-
    \+ checkBorders(Line, Column),
    getElemsAdjacent(Board, Line, Column, Elems),
    verifyConnection(Board, Line, Column, Elems),
    testBoardConnection(Board, T).

testBoardConnection(_, []).

verifyConnection(Board, Line, Column, Elems):-
    getElement(Board, Line, Column, P),
    getElemsAround(Board, Line, Column, Around),
    sum(Around,#=,A),
    sum(Elems,#=,H),
    ((P#=0 #/\ H#=<3 #/\ A#=<7) #\/ (P#=1 #/\ H#=2)),
    preventCrossing(P, Board,Line, Column).
```

- e por fim aplicamos a seguinte restrição para que a cobra não se toque na diagonal: - para uma determinada posição (a 1) ter uma peça a 1 na diagonal, ambas as posições têm de ter uma e só uma casa adjacente a 1 em comum.

```
% Prevents the snake from touching itself on the diagonal
% For 2 pieces of the snake to touch each other, they must
% have one and only one adjacent piece in common
% preventCrossing(+reference_element, +board, +line, +column)
preventCrossing(P, Board, Line, Column):-
    findall(L-C, (adjacent(Line, Column, L, C), nth0(L,Board,Temp), nth0(C,Temp, _)) , Adj),
```

```
7       findall(L1-C1, diagonal(Line, Column, L1, C1), Diag),
8       findall(LL-CC, (member(LL-CC, Diag), nth0(LL,Board,Temp), nth0(CC, Temp, _)), Diags),
9       preventCrossing1(P, Board, Adj, Diags).
10
11  preventCrossing1(P,Board, Adj, [L-_|T]):-
12      \+ nth0(L, Board, _),
13      preventCrossing1(P,Board, Adj, T).
14
15  preventCrossing1(P,Board, Adj, [L-C|T]):-
16      nth0(L, Board, Temp),
17      \+ nth0(C, Temp, _),
18      preventCrossing1(P, Board, Adj, T).
19
20  preventCrossing1(P,Board, Adj, [L-C|T]):-
21      findall(L1-C1, (adjacent(L, C, L1, C1),nth0(L1,Board, Temp),nth0(C1, Temp, _)), D_Adj),
22      findall(Line-Col, (member(Line-Col, Adj), member(Line-Col,D_Adj)), Interset1),
23      getElem(Board, Interset1, Interset),
24      nth0(L, Board, Temp),
25      nth0(C, Temp, Elem),
26      sum(Interset, #=, Sum),
27      ((Sum#=0 #/\ P#=1) #=> Elem#=0),
28      preventCrossing1(P,Board, Adj, T).
29
30  preventCrossing1(_,_, _, []).
```

3.3 Estratégia de Pesquisa

De modo a tornar a pesquisa mais eficiente, foi usado no predicado labeling, a opção ffc - first fail constraint. Isto faz com que seja usada a restrição mais rápida: é escolhida a variável com o domínio mais pequeno, com menos restrições e mais à esquerda.

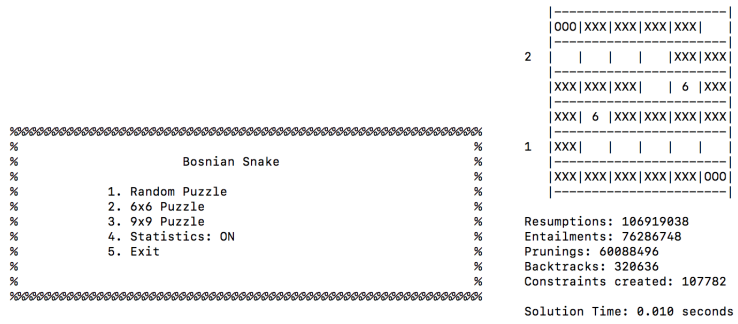
4 Visualização da Solução

Para a visualização da solução foi criados os métodos *printMatrix*, *printRowMatrix* e *printHorizontalDivision* que tratam da representação da matriz e seus limites e da representação da cobra(cabeça,cauda e corpo).

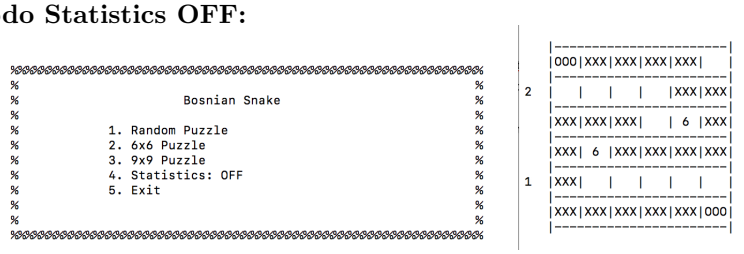
Foi criado o método *printStatisticts* que quando a opção *Statistics* estiver ativa imprime, em conjunto com a matriz, o número de passos feitos na resolução do problema. Para além deste método existe ainda o método *print_time* que imprime o tempo que demorou a resolução do problema.

Abaixo podemos observar o funcionamento desta funcionalidade, considerando o mesmo problema de 6x6:

Modo Statistics ON:



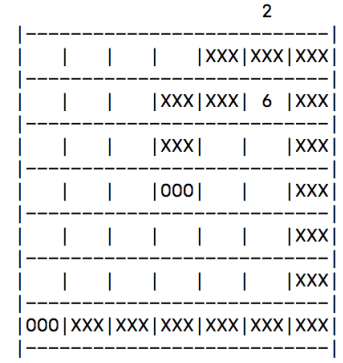
Modo Statistics OFF:



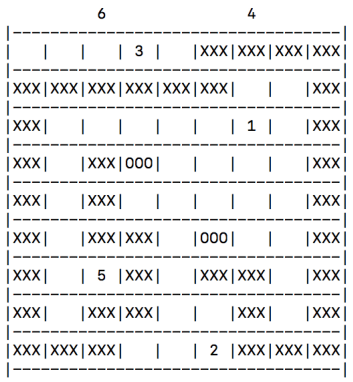
5 Resultados

Abaixo podemos ver alguns exemplos de problema de diferentes complexidades resolvidos onde podemos ver as restrições às casas envolventes e as restrições às linhas/colunas serem respeitadas.

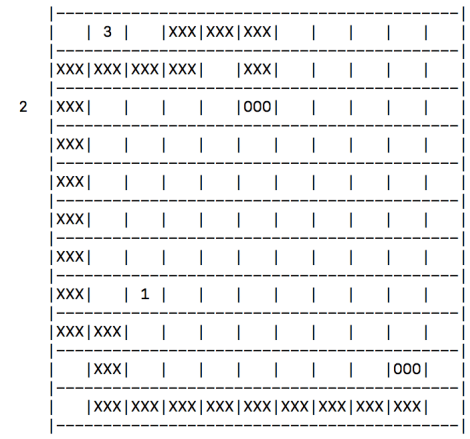
Relativamente a tempos de execução, todos os problemas depois de criados foram rapidamente resolvidos pelo programa, sendo o maior fator de demoras a criação do problema em si.De forma geral quanto mais complexo o problema maior o número de passos dados, não sendo isto, como já foi dito, um grande obstáculo a uma rápida execução da resolução.



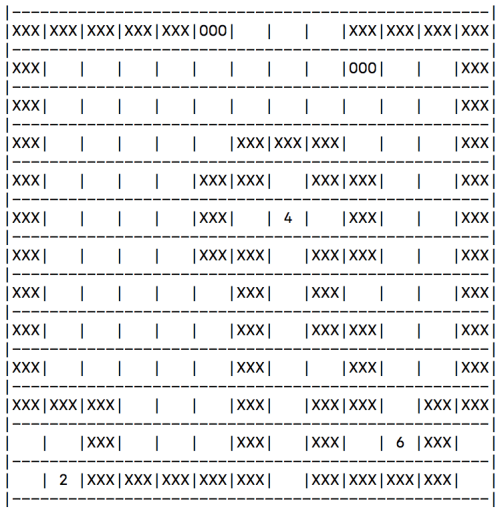
Puzzle 7x7



Puzzle 9x9



Puzzle 11x11



Puzzle 13x13

6 Conclusão

Os objetivos para este projeto foram inteiramente atingido, quer a resolução do problema tendo este diferentes complexidades, bem como a implementação de um método de geração dinâmica de problemas.

A realização deste projeto permitiu aos elementos do grupo uma melhor precessão sobre a linguagem prolog,mais precisamente sobre restrições em prolog e a sua grande utilidade na resolução de problemas de decisão e otimização.

Apesar da difuldade em realizar algumas tarefas simples, prolog é muito útil devido a uma maior facilidade na resolução de questões complexas,em relação a outras linguagens.

7 Referências

[1] Walker, Anderson : Shading and Loops. Ep-2. Bosnian Snake, *http://logicmastersindia.com/lmitests/dl.asp?attachmentid=645&view=1*, 2 (2017)

[2] SICStus Prolog: *https://sicstus.sics.se/*

8 Anexos

8.1 Código Fonte

8.1.1 bosnian.pl

```
:-use_module(library(lists)).
:-use_module(library(clpfd)).
:-use_module(library(statistics)).
:-use_module(library(random)).
:-use_module(library(timeout)).
:-include('interface.pl').
:-include('logic.pl').
:-include('utils.pl').
:-include('generator.pl').


start:-
    mainMenu(0),
    !.

mainMenu(EnableV):-
    clearScreen,
    write('    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%'),nl,
    write('    %                                                             %'),nl,
    write('    %                      Bosnian Snake                          %'),nl,
    write('    %                                                             %'),nl,
    write('    %            1. Random Puzzle                                 %'),nl,
    write('    %            2. 6x6 Puzzle                                    %'),nl,
    write('    %            3. 9x9 Puzzle                                    %'),nl,
    printStatistictsStatus(EnableV),
    write('    %            5. Exit                                          %'),nl,
    write('    %                                                             %'),nl,
    write('    %                                                             %'),nl,
    write('    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%'),nl,
    nl,nl,
    write('Please choose an option: '),
    read(R),
    menu(R, EnableV).

% ----------------------------------------------------------------------
% Menu 1 - RANDOM PUZZLE
% ----------------------------------------------------------------------

menu(X, Stats):-
    X==1,
    clearScreen,
    generator,
    !.

% ----------------------------------------------------------------------
% Menu 2 - 6X6 PUZZLE
```

```
% ------------------------------------------------------------------------

menu(X, Stats):-
    X==2,
    clearScreen,
    asserta(sumLine(1,2)),
    asserta(sumLine(4,1)),
    asserta(sumAround(2,4,6)),
    asserta(sumAround(3,1,6)),
    asserta(snakeHead(0,0)),
    asserta(snakeTail(5,5)),
    solver(6, Stats),
    cleanDynamicStuff.

% ------------------------------------------------------------------------
% Menu 3 - 9X9 PUZZLE
% ------------------------------------------------------------------------

menu(X,Stats):-
    X==3,
    clearScreen,
    asserta(snakeHead(3,3)),
    asserta(snakeTail(5,5)),
    asserta(sumCol(6,4)),
    asserta(sumCol(2,6)),
    asserta(sumAround(2,6,1)),
    asserta(sumAround(0,3,3)),
    asserta(sumAround(6,2,5)),
    asserta(sumAround(8,5,2)),
    solver(9,Stats),
    cleanDynamicStuff.

% ------------------------------------------------------------------------
% Menu 4 - STATISTICS
% ------------------------------------------------------------------------

menu(X, EnableV):-
    X==4,
    EnableV1 is mod(EnableV+1,2),
    mainMenu(EnableV1).

% ------------------------------------------------------------------------
% Exit GAME
% ------------------------------------------------------------------------

menu(X, _):-
    X==5,
    true.

% ------------------------------------------------------------------------
% Invalid Option
```

```
% ------------------------------------------------------------------------
menu(_, _):-
    mainMenu(_).

% Print statistics status
printStatistictsStatus(0):-
    write('    %            4. Statistics: OFF                          %'),nl.

printStatistictsStatus(1):-
    write('    %            4. Statistics: ON                           %'),nl.

% Print statistics
printStatisticts(1):-
    fd_statistics.

printStatisticts(_).

% Solves the problem and displays the solution
solver(Size, Statistics):-
    solveProb(Size, Board),
    printMatrix(Board),nl,
    printStatisticts(Statistics),
    print_time.
```

8.1.2 logic.pl

```
% Sum of line
% sumLine(+n,+sum)
:-dynamic sumLine/2.

% Sum of column
% sumCol(+n,+sum)
:-dynamic sumCol/2.

% Sum of neighbours
% sumAround(+line, +column, ?sum)
:-dynamic sumAround/3.

% Snake Head
% snakeHead(-line, -column)
:-dynamic snakeHead/2.

% Snake Tail
% snakeTail(-line, -column)
:-dynamic snakeTail/2.

%Returns all elements of a specific line
%  getElemsLine(+Board, +Line, -Elems)
getElemsLine(Board, Line, Elems):-
    nth0(Line, Board, Elems).

%Returns all elements of a specific column
```

```
%  getElemsColumn(+Board, +Column, -Elems)
getElemsColumn(Board, Column, Elems):-
    transpose(Board, New_Board),
    nth0(Column, New_Board, Elems).

%Returns all elements of a specific column
%  getElemsAround(+Board, +Line, +Column, -Elems)
getElemsAround(Board, L, C, Elems):-
    findall(L1-C1, neighbor(L, C, L1, C1), Temp),
    getElem(Board, Temp, Elems).

% Returns all valid elements of a list of pairs
% getElem(+board, +List, +Elems)
getElem(Board, [Line-Column|T], [Head|Tail]):-
    nth0(Line, Board, Tmp),
    nth0(Column, Tmp, Head),
    getElem(Board, T, Tail).

getElem(Board, [_|T], Elems):-
    getElem(Board, T, Elems).

getElem(_, [], []).

% Return the neighbor of a specific position L-C
% neighbor(+line, +column, -neighbor_line, -neighbor_column)
neighbor(L,C, L1, C1):-
    L1 is L-1, C1 is C-1.

neighbor(L,C, L1, C1):-
    L1 is L-1,C1 is C.

neighbor(L,C, L1, C1):-
    L1 is L-1,C1 is C+1.

neighbor(L,C, L1, C1):-
    L1 is L,C1 is C-1.

neighbor(L,C, L1, C1):-
    L1 is L,C1 is C+1.

neighbor(L,C, L1, C1):-
    L1 is L+1,C1 is C-1.

neighbor(L,C, L1, C1):-
    L1 is L+1,C1 is C.

neighbor(L,C, L1, C1):-
    L1 is L+1,C1 is C+1.

% Test all horizontal restrictions of the board
% testHorizontalSum(+ board)
testHorizontalSum(Board):-
```

```
    findall(Line-Sum, sumLine(Line,Sum),All_Res),
    testHorizontalSum(Board, All_Res).

testHorizontalSum(Board, [Line-Sum|T]):-
    getElemsLine(Board, Line, Elems),
    sum(Elems, #=, Sum),
    testHorizontalSum(Board, T).

testHorizontalSum(_, []).

% Test all vertical restrictions of the board
% testHorizontalSum(+ board)
testVerticalSum(Board):-
    findall(Column-Sum, sumCol(Column,Sum),All_Res),
    testVerticalSum(Board, All_Res).

testVerticalSum(Board, [Column-Sum|T]):-
    getElemsColumn(Board, Column, Elems),
    sum(Elems, #=, Sum),
    testVerticalSum(Board, T).

testVerticalSum(_, []).

% Test all around position restrictions of the board
% testHorizontalSum(+ board)
testNeighboursSum(Board):-
    findall([Line, Column,Sum], sumAround(Line,Column,Sum),All_Res),
    testNeighboursSum(Board, All_Res).

testNeighboursSum(Board, [Head|T]):-
    nth0(0, Head, Line),
    nth0(1, Head, Column),
    nth0(2, Head, Sum),
    nth0(Line, Board, Temp),
    nth0(Column, Temp, Position),
    Position #= 0,
    getElemsAround(Board, Line, Column, Elems),
    sum(Elems, #=, Sum),
    testNeighboursSum(Board, T).

testNeighboursSum(_, []).

% Return the position adjacent of a specific position L-C
% adjacent(+line, +column, -adjacent_line, -adjacent_column)
adjacent(Line,Col,L1,C1):-
    L1 is Line-1,
    C1 is Col.

adjacent(Line,Col,L1,C1):-
    L1 is Line+1,
    C1 is Col.
```

```
adjacent(Line,Col,L1,C1):-
    L1 is Line,
    C1 is Col-1.

adjacent(Line,Col,L1,C1):-
    L1 is Line,
    C1 is Col+1.

% Return the diagonal of a specific position L-C
% diagonal(+line, +column, -diagonal_line, -diagonal_column)
diagonal(Line,Col,L1,C1):-
    L1 is Line-1,
    C1 is Col-1.

diagonal(Line,Col,L1,C1):-
    L1 is Line+1,
    C1 is Col+1.

diagonal(Line,Col,L1,C1):-
    L1 is Line-1,
    C1 is Col+1.

diagonal(Line,Col,L1,C1):-
    L1 is Line+1,
    C1 is Col-1.

% Return the diagonal adjacent of a specific position L-C
% diagonal(+line, +column, -diagonal_line, -diagonal_column)
getElemsAdjacent(Board, Line, Column, Elems):-
    findall(L1-C1, adjacent(Line, Column, L1, C1), Temp),
    getElem(Board, Temp, Elems).


%Check if a position is the tail or the head of the snake
% checkBorders(+line,+column)
checkBorders(L, C):-
    snakeHead(L,C);
    snakeTail(L,C).

% Teste the board connectivity
% All pieces of the snake must have 2 adjacente pieces, except the borders
% that must have only 1
% testBoardConnection(+ board)
testBoardConnection(Board):-
    findall(Line-Column, (nth0(Line, Board, Tmp),nth0(Column, Tmp, _)), Indexes),
    testBoardConnection(Board, Indexes).

testBoardConnection(Board, [Line-Column|T]):-
    checkBorders(Line, Column),
    getElemsAdjacent(Board, Line, Column, Elems),
    sum(Elems, #=, 1), % only has 1 connection
    testBoardConnection(Board, T).
```

```
testBoardConnection(Board, [Line-Column|T]):-
    \+ checkBorders(Line, Column),
    getElemsAdjacent(Board, Line, Column, Elems),
    verifyConnection(Board, Line, Column, Elems),
    testBoardConnection(Board, T).

testBoardConnection(_, []).

verifyConnection(Board, Line, Column, Elems):-
    getElement(Board, Line, Column, P),
    getElemsAround(Board, Line, Column, Around),
    sum(Around,#=,A),
    sum(Elems,#=,H),
    ((P#=0 #/\ H#=<3 #/\ A#=<7) #\/ (P#=1 #/\ H#=2)),
    preventCrossing(P, Board,Line, Column).

% Returns the element of a position
% getElement(+ board, +line, +column, -element)
getElement(Board, Line, Column, Elem):-
     nth0(Line,Board,Temp),
     nth0(Column,Temp, Elem).

% Returns true if an element is not a member of a list
% notMember(+element, +list)
notMember(A, List):-
    \+ member(A, List).

% Prevents the snake from touching itself on the diagonal
% For 2 pieces of the snake to touch each other, they must
% have one and only one adjacent piece in common
% preventCrossing(+reference_element, +board, +line, +column)
preventCrossing(P, Board, Line, Column):-
    findall(L-C, (adjacent(Line, Column, L, C), nth0(L,Board,Temp), nth0(C,Temp, _)) , Adj
    findall(L1-C1, diagonal(Line, Column, L1, C1), Diag),
    findall(LL-CC, (member(LL-CC, Diag), nth0(LL,Board,Temp), nth0(CC, Temp, _)), Diags),
    preventCrossing1(P, Board, Adj, Diags).

preventCrossing1(P,Board, Adj, [L-_|T]):-
    \+ nth0(L, Board, _),
    preventCrossing1(P,Board, Adj, T).

preventCrossing1(P,Board, Adj, [L-C|T]):-
    nth0(L, Board, Temp),
    \+ nth0(C, Temp, _),
    preventCrossing1(P, Board, Adj, T).

preventCrossing1(P,Board, Adj, [L-C|T]):-
    findall(L1-C1, (adjacent(L, C, L1, C1),nth0(L1,Board, Temp),nth0(C1, Temp, _)), D_Adj)
    findall(Line-Col, (member(Line-Col, Adj), member(Line-Col,D_Adj)), Interset1),
    getElem(Board, Interset1, Interset),
    nth0(L, Board, Temp),
```

```
    nth0(C, Temp, Elem),
    sum(Interset, #=, Sum),
    ((Sum#=0 #/\ P#=1) #=> Elem#=0),
    preventCrossing1(P,Board, Adj, T).

preventCrossing1(_,_, _, []).


% Solves the problem
% solveProb(+ board_size, -solution_board)
solveProb(Size, Board):-
    reset_timer,
    Size1 is Size*Size,

    length(List, Size1),
    domain(List, 0,1),

    % ADD SNAKE HEAD %
    snakeHead(Head_L, Head_C),
    calculateIndex(Size, Head_L, Head_C, Index),
    setElemByIndex(List,Index, 1),

    % ADD SNAKE TAIL %
    snakeTail(Tail_L, Tail_C),
    calculateIndex(Size, Tail_L, Tail_C, Index1),
    setElemByIndex(List,Index1, 1),

    list_to_matrix(List, Size, Board),

    testHorizontalSum(Board),
    testVerticalSum(Board),
    testNeighboursSum(Board),
    testBoardConnection(Board),

    labeling([ffc], List).
```

8.1.3 interface.pl

```
printMatrix(Board):-
    printMatrix(Board,-1,-1).

printMatrix(Board,-1,_):-
    printRowMatrix(Board, -1, -1),
    nl,
    length(Board, Size),
    printHorizontalDivision(Size),
    printMatrix(Board, 0, -1).


printMatrix([Head|Tail],Line,Column):-
    printRowMatrix(Head, Line, Column),
    write('|'),nl,
```

```
    length(Head, Size),
    printHorizontalDivision(Size),
    Line1 is Line+1,
    printMatrix(Tail, Line1, -1).

printMatrix([],_,_).

printRowMatrix([],_,_).

printRowMatrix(Board, Line,-1):-
    sumLine(Line,Sum),
    Sum < 10,
    write(Sum),
    write('  '),
    printRowMatrix(Board, Line, 0).

printRowMatrix(Board, Line,-1):-
    sumLine(Line,Sum),
    write(Sum),
    write('   '),
    printRowMatrix(Board, Line, 0).

printRowMatrix(Board, Line,-1):-
    write('   '),
    printRowMatrix(Board, Line, 0).

printRowMatrix([_|Tail],Line,Column):-
    checkBorders(Line,Column),
    write('|OOO'),
    C1 is Column+1,
    printRowMatrix(Tail, Line, C1).

printRowMatrix([_|Tail],Line,Column):-
    sumAround(Line, Column, Sum),
    write('| '),
    write(Sum),
    write(' '),
    C1 is Column+1,
    printRowMatrix(Tail, Line, C1).

printRowMatrix([Head|Tail],Line,Column):-
    Head = 0,
    write('|   '),
    C1 is Column+1,
    printRowMatrix(Tail, Line, C1).

printRowMatrix([Head|Tail],Line,Column):-
    Head = 1,
    write('|XXX'),
    C1 is Column+1,
    printRowMatrix(Tail, Line, C1).
```

```
printRowMatrix([_|Tail],-1,Column):-
    sumCol(Column,Sum),
    write('  '),
    write(Sum),
    write(' '),
    C1 is Column+1,
    printRowMatrix(Tail,-1, C1).

printRowMatrix([_|Tail],-1,Column):-
    write('    '),
    C1 is Column+1,
    printRowMatrix(Tail,-1, C1).

printRowMatrix(Board,-1,-1):-
    write(' '),
    printRowMatrix(Board,-1, 0).

printHorizontalDivision(Length):-
    write('   |'),
    Length1 is Length*4,
    printHorizontalDivision(Length1, 1).

printHorizontalDivision(Length, Length):-
    write('|'),nl.

printHorizontalDivision(Length, I):-
    write('-'),
    I1 is I+1,
    printHorizontalDivision(Length, I1).
```

8.1.4 generator.pl

```
generator:-
    write('Generating ...'),nl,
    repeat,
        cleanDynamicStuff,
        write('Solving ...'),nl,
        random(5,20, Size),
        randomSnakeBorders(Size),
        randomColumnRestrictions(Size),
        randomLineRestrictions(Size),
        randomAroundRestrictions(Size),
        time_out(solveProb(Size, Board),1000,Result),
        write(Result),nl,
        !,

    (Result = time_out -> generator; printMatrix(Board),cleanDynamicStuff, true).


cleanDynamicStuff:-
    retractall(sumLine(_,_)),
    retractall(sumCol(_,_)),
```

```
    retractall(sumAround(_,_,_)),
    retractall(snakeHead(_,_)),
    retractall(snakeTail(_,_)).



randomSnakeBorders(Size):-
    random(0, Size, L1),
    random(0, Size, C1),
    random(0, Size, L2),
    random(0, Size, C2),
    asserta(snakeHead(L1,C1)),
    asserta(snakeTail(L2,C2)).

randomColumnRestrictions(Size):-
    Max is integer(Size/3),
    random(0,Max, N),
    randomColumnRestrictions(Size, N,0).

randomColumnRestrictions(_,Total, Total).

randomColumnRestrictions(Size,Total, I):-
    repeat,
        Max is integer(Size/2),
        random(0,Size, Col),
        \+ sumCol(Col,_),
        random(1,Max, Sum),
        asserta(sumCol(Col,Sum)),
        !,
    I1 is I+1,
    randomColumnRestrictions(Size,Total, I1).

randomLineRestrictions(Size):-
    Max is integer(Size/3),
    random(0,Max, N),
    randomLineRestrictions(Size, N,0).

randomLineRestrictions(_,Total, Total).

randomLineRestrictions(Size,Total, I):-
    repeat,
        Max is integer(Size/2),
        random(0,Size, Line),
        \+ sumLine(Line,_),
        random(1,Max, Sum),
        asserta(sumLine(Line,Sum)),
        !,
    I1 is I+1,
    randomLineRestrictions(Size,Total, I1).

randomAroundRestrictions(Size):-
    Max is integer(Size/3),
```

```
    random(0,Max, N),
    randomAroundRestrictions(Size, N,0).

randomAroundRestrictions(_,Total, Total).

randomAroundRestrictions(Size,Total, I):-
    repeat,
        Max is 7,
        random(0,Size, Line),
        random(0,Size, Col),
        \+ sumAround(Line,Col,_),
        random(0,Max, Sum),
        asserta(sumAround(Line,Col, Sum)),
        !,
    I1 is I+1,
    randomAroundRestrictions(Size,Total, I1).
```

8.1.5 utils.pl

```
% clean the Screen with 50 new lines
clearScreen :-
        newLine(50), !.

% display new lines
% newLine(+ Number)
newLine(Number) :-
        newLine(0, Number).

% display new lines
% newLine(+ Line,  + Limit)
newLine(Line, Limit) :-
        Line < Limit,
        LineInc is Line + 1,
        nl,
        newLine(LineInc, Limit).

newLine(_,_).

% convert an ascii code to a decimal number
% ascii_to_dec(+ N, - N1)
ascii_to_dec(N,N1):-
        N1 is N-48.

% reset the timer
reset_timer:-
        statistics(walltime,_).

% Print the elapsed time
print_time :-
        statistics(walltime, [_, ElapsedTime | _]),
        nl,
        format('Solution Time: ~3d seconds',ElapsedTime), nl, nl.
```

```
% Convert a list to matrix
% list_to_matrix(+list, +matrix_size, -matrix)
list_to_matrix([], _, []).

list_to_matrix(List, Size, [Row|Matrix]):-
        list_to_matrix_row(List, Size, Row, Tail),
        list_to_matrix(Tail, Size, Matrix).

list_to_matrix_row(Tail, 0, [], Tail).

list_to_matrix_row([Item|List], Size, [Item|Row], Tail):-
        NSize is Size-1,
        list_to_matrix_row(List, NSize, Row, Tail).

calculateIndex(Size, Line, Column, Index):-
        Index is (Size * Line + Column).

calculateLineColumn(Size, Index, Line, Column):-
        Line is integer(Index/Size),
        Column is mod(Index,Size).

setElemByIndex(List, Index, Elem):-
        setElemByIndex(List, Index, 0, Elem).

setElemByIndex([Head|_], Index, Index, Elem):-
        Head = Elem.

setElemByIndex([_|Tail], Index, I, Elem):-
        I1 is I+1,
        setElemByIndex(Tail, Index, I1, Elem).

intersection([], _, []).

intersection([H1|T1], L2, [H1|Res]) :-
        member(H1, L2),
        intersection(T1, L2, Res).

intersection([_|T1], L2, Res) :-
        intersection(T1, L2, Res).


disjunction([], _,[]).

disjunction([H1|T1], L2, [H1|Res]) :-
        disjunction(T1, L2, Res).

disjunction([H1|T1], L2, Res) :-
        member(H1, L2),
        disjunction(T1, L2, Res).
```