Resolução de Problema de Decisão usando Programação em Lógica com Restrições - Bosnian Snake

Mariana Lopes Silva, Francisca Leão Cerquinho Ribeiro da Fonseca

Faculdade de Engenharia da Universidade do Porto
Roberto Frias, sn, 4200-465 Porto, Portugal
FEUP-PLOG, Turma 3MIEIC02, Grupo Bosnian Snake_4

Resumo Este artigo complementa o segundo projecto da Unidade Curricular de Programação em Logica, do Mestrado Integrado em Engenharia Informática e Computação. O objetivo deste trabalho, implementado através da linguagem de programação em lógica, Prolog, passa por resolver um problema de decisão com restrições, nomeadamente o *puzzle Bosnian Snake*, que tem como principal objetivo encontrar um caminho contínuo e único cujas posições inicial e final são dadas, respeitando certas restrições, nomeadamente a imposição do número, indicado no tabuleiro, de células pintadas quer nas linhas quer nas colunas.
O problema proposto foi na sua totalidade implementado, através da utilização dos predicados disponibilizados pelo *SICStus Prolog*, onde foi tido em conta não só a funcionalidade do jogo, como também a eficiência do próprio código. Assim, foi possível a consolidação dos conceitos lecionados ao longo da unidade curricular.

Keywords: bosnian snake, sicstus, prolog, feup

1 Introdução

No âmbito da unidade curricular Programação em Lógica foi-nos proposto a realização de um problema em lógica com restrições *Prolog*, pondo à prova os nossos conhecimentos relativamente a regras e problemas intrínsecos à linguagem.

Efetivamente, ao longo deste relatório irão ser abordados três grandes tópicos, nomeadamente, a descrição do problema, a sua abordagem e solução. Teremos em conta, também, no final, as principais conclusões deste projeto.

Desta forma, procuramos responder ao que nos é exigido, de forma sucinta e explícita, sendo nossa intenção fazer com que o presente relatório sirva de guia e suporte para os interessados na resolução do *puzzle*.

2 Descrição do problema

O problema do *puzzle Bosnian Snake* consiste em descobrir um caminho de uma cobra, contínuo e único de 1 célula, cuja cabeça e cauda são dados. A

acrescentar, a cobra não se toca, mesmo na diagonal e são referenciadas números no interior e exterior do tabuleiro que restringem o número de células pintadas. Por um lado, os números representados fora do tabuleiro indicam o número de células que podem estar pintadas nessa(s) linha(s) e/ou coluna(s). Por outro lado, os números representados no interior de uma célula do tabuleiro indicam quantas das oito células a seu redor podem estar pintadas.

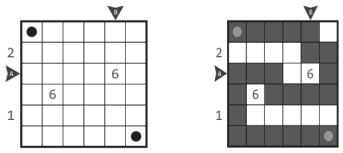

Figura 1: Exemplo de um Puzzle Bosnian Snake resolvido

3 Abordagem

A Programação em Lógica com Restrições – PLR, ou CLP (*Constraint Logic Programming*) é uma classe de linguagens de programação combinando declaratividade da programação em lógica e eficiência da resolução de restrições. As principais vantagens da sua utilização é o reduzido tempo de desenvolvimento, a facilidade de manutenção, a eficiência na resolução e a clareza e brevidade dos programas.

A primeira etapa na abordagem do problema foi como modelar o *puzzle* como um problema de restrições, analisando quais as variáveis de decisão a utilizar no predicado de *labeling*, bem como, o seu domínio.

3.1 Variáveis de decisão

A solução pretendida para este *puzzle* é o próprio tabuleiro com o caminho da cobra representado. Desta forma, neste problema a variável de decisão (ou variável de domínio) utilizada no predicado *labeling* foi uma lista de listas, denominada *puz*, cujo domínio são os valores 0 e 1. Sendo 0 uma célula não pintada e 1 uma célula pintada.

3.2 Restrições

Considera-se, e até ao final deste artigo, que ***NumberOut*** é o número representado fora do tabuleiro que indica o número de células que têm de estar colocadas a 1 em cada linha ou coluna e ***NumberIn*** é o número representado no interior de qualquer uma das células que indica quantas das oito células a seu redor têm de estar colocadas a 1.

Para a resolução deste problema foram criadas várias restrições rígidas, nomeadamente:

1. Cada coluna ou linha que contenha o ***NumberOut*** deve possuir exatamente esse número de células pintadas;

2. Cada célula que contenha o ***NumberIn*** deve possuir exatamente esse número de células à sua volta pintadas;

3. A impossibilidade de haver células pintadas adjacentes;

4. A conetividade do caminho.

Restrição imposta pelo *NumberOut*

Para um tabuleiro de tamanho ***Size*** e dado o ***NumberOut***, bem como a linha (***Row***) ou a coluna (***Col***), onde a restrição será aplicada, garante-se, usando o predicado *global_cardinality*, que o número de ocorrências de células a 1 nessa linha ou coluna é igual a ***NumberOut***. Os predicados responsáveis por garantir esta restrição são os *cellsOfRestrictionOut_ROW(List,Number,Row,Size)*, que está presente nas linhas e *cellsOfRestrictionOut_COL(List,Number,Col,Size)*, que está presente nas linhas, do Anexo I.

Restrição imposta pelo *NumberIn*

Esta restrição é muito semelhante à anterior, na medida em que temos que garantir que o número de ocorrências de células a 1 tem de ser exatamente igual a ***NumberIn***. Desta vez, para um dado tabuleiro de tamanho **Size** e dado o número da linha (***Nrow***) e número da coluna (***Ncol***) de uma determinada célula, o número de células à volta dessa, colocadas a 1, tem de ser exatamente igual a ***NumberIn***, usando o predicado *global_cardinality*. Para isso, é necessário obter a lista de células vizinhas a uma determinada célula. Se a célula em questão for um dos quatro cantos, o número de células vizinhas é 3, para as células dos extremos direito, esquerdo, superior e inferior é 5 e para as células centrais é 8. O predicado responsável por garantir esta restrição é o *cellsAround(List, Nrow, Ncol, NumberIn, Size)*, que está presente nas linhas, do Anexo I.

Conetividade do caminho e Restrição das células pintadas não estarem adjacentes

Para garantir simultanemanete a conetividade do caminho e o facto de células pintadas não puderem estar adjacentes, foi criado o predicado *imposeConectivity([_|Tail],List,Dim,Position)* que percorre todas as células do tabuleiro, e caso a célula em questão esteja a 1, é garantido que:

- Para os cantos superior direito e inferior esquerdo que a soma das células vizinhas situadas a norte, sul, este e oeste tem de ser exatamente dois e a soma das três células a seu redor tem de ser exatamente dois;

- Para os cantos superior esquerdo e inferior direito que a soma das células vizinhas situadas a norte, sul, este e oeste tem de ser exatamente um e a soma das três células a seu redor tem de ser menor ou igual a dois;

- Para as restantes células, impõe-se que a soma das células vizinhas situadas a norte, sul, este e oeste tem de ser exatamente dois e a soma das oito células a seu redor tem de ser menor ou igual a quatro.

3.3 Função de avaliação

Para este problema, como a solução é verificável através da visualização do tabuleiro, não foi necessário fazer uma avaliação da solução obtida.

3.4 Estratégia de pesquisa

A estratégia de etiquetagem (labeling) utilizada foi a *default*, uma vez que foram testadas várias estratégias, nomeadamente no que diz respeito à ordenação de valores, como por exemplo, o down (domínio explorado por ordem descendente) e ordenação de variáveis, como o occurrence (mais restrições suspensas, mais à esquerda) e ffc (menor domínio, mais restrições suspensas), mas nenhuma das anteriores obteve uma solução melhor, como pode ser observado pela figura 5.

4 Visualização da Solução

Os predicados responsáveis pela visualização da solução encontram-se no ficheiro *board.pl*, que se encontra nos anexos deste artigo. Através destes predicados, o utilizador consegue visualizar as pistas interiores e exteriores do *puzzle* e o caminho correto da cobra desde a posição inicial à final, representado por ***. Além disso, é possível gerar um tabuleiro aleatório através do predicado *randomPuzzle*.

Seguem-se exemplos da resolução de diversos *puzzle's*.

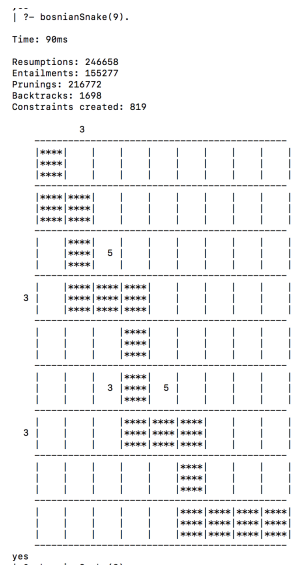

Figura 2: Resultado de um *puzzle* 9x9 com três pistas interiores e exteriores

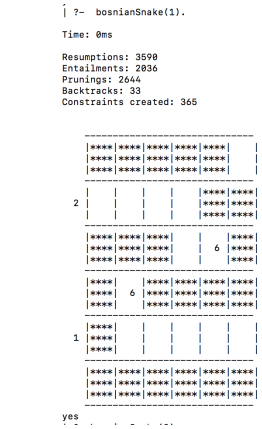

Figura 3: Resultado de um *puzzle* 6x6 com duas pistas interiores e exteriores

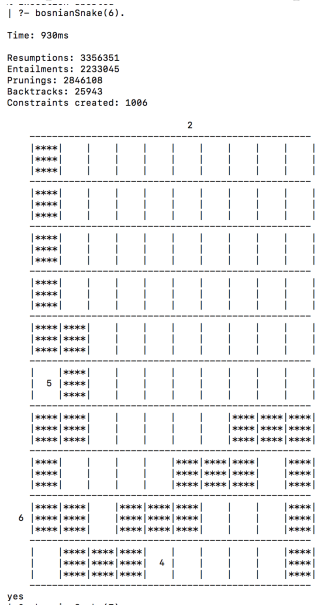

Figura 4: Resultado de um *puzzle* 12x12 com duas pistas interiores e exteriores

5 Resultados

De forma a analisar os dados estatísticos obtidos para cada tipo de tabuleiro e para cada uma das três estratégias de etiquetagem diferentes, recolhemos alguns dados e representamo-los graficamente.

A partir dos gráficos, podemos concluir que a melhor estratégia de *labeling* é a *default*, uma vez que para o mesmo número de restrições e para a mesma

dimensão do tabuleiro, consegue resolver a solução num espaço de tempo mais curto. A *ffc* será a segunda melhor, seguida da *occurrence* e *down*.

				C/ FFC		C/OCCURRENCE		C/DOWN	
Nr puzzle	Dim	Tempo(ms)	Backtracks	Tempo(ms)	Backtracks	Tempo(ms)	Backtracks	Tempo(ms)	Backtracks
1	6x6	0	33	10	21	10	31	20	33
2	12x12	181860	3355738	2768360	37553639	1861070	37763067	195040	3480311
3	8x8	10	65	10	40	10	13951	10	73
4	7x7	20	266	50	632	80	632	50	457
5	15x15	-	-	-	-	-	-	-	-
6	10x10	0	14	1770	85633	3210	34633	26130	234912
7	5x5	930	25943	0	12	0	12	0	14
8	3x3	0	1	0	1	0	1	0	1
9	11x11	90	1698	100	1117	140	1117	420	3516

Figura 5: Tempo decorrido para cada tipo de tabuleiro e com diferentes estratégias de etiquetagem. Celulas com hífen signicam que não há solução para a dimensão e restrições do tabuleiro e respetivas restrições.

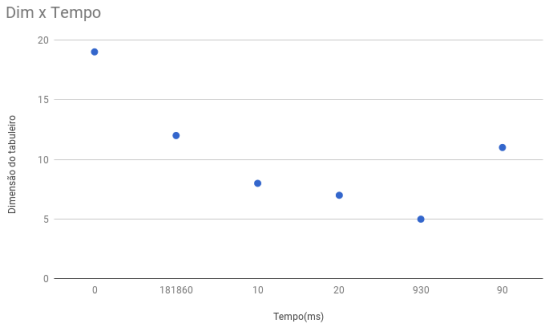

Figura 6: Tempo decorrido para cada tipo de tabuleiro com a estratégia de etiquetagem escolhida (*default*).

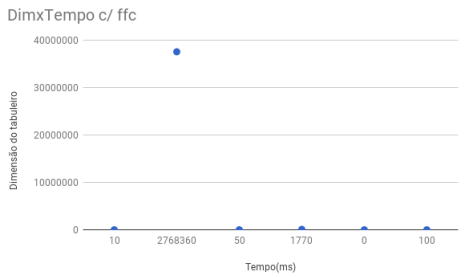

Figura 7: Tempo decorrido para cada tipo de tabuleiro com a estratégia de etiquetagem *ffc*

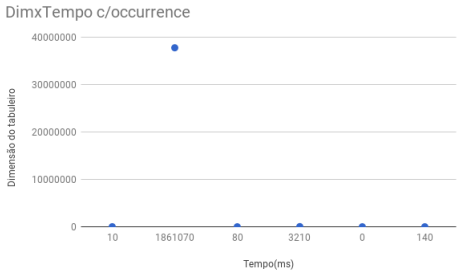

Figura 8: Tempo decorrido para cada tipo de tabuleiro com a estratégia de etiquetagem *occurrence*

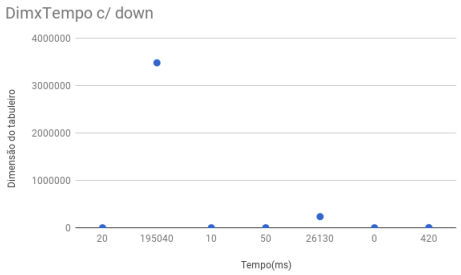

Figura 9: Tempo decorrido para cada tipo de tabuleiro com a estratégia de etiquetagem *down*

6 Conclusões e Trabalho Futuro

A realização deste projeto permitiu-nos perceber as notórias e múltiplas vantagens da linguagem *Prolog*, nomeadamente a ampla variedade questões de decisão e otimização que os módulos de restrições englobam.

Efetivamente, a solução implementada correspondeu ao que era exigido, tendo sido cumpridas todas as metas iniciais. Os resultados obtidos foram os esperados e os dados estatísticos permitiram-nos concluir acerca da melhor estratégia de etiquetagem a utilizar.

Em suma, este trabalho teve uma grande influência no nosso percurso como alunas de Engenharia Informática e Computação, pois permitiu-nos não só solidificar os conhecimentos lecionados, como também pôr em prática a construção de predicados baseados na Programação em Lógica com Restrições.

Referências

1. Clocksin, W. F.; Programming in prolog. ISBN: 0-387-58350-5

7 Anexos

Anexo I

Ficheiro "bosnianSnake.pl"

```
:- use_module(library(random)).
:- include('conectivity.pl').
:- include('board.pl').

puz(1, [1-1, 6-6], 6-6, [2-2,5-1], [], [3-5-6, 4-2-6]).
puz(2, [1-1, 12-12], 12-12, [], [7-6,12-4], [6-4-7, 4-8-5]).
puz(3, [1-1, 8-8], 8-8, [2-4], [5-1], [3-5-6, 4-2-7]).
puz(4, [1-1, 7-7], 7-7, [5-3], [3-1], [4-2-3, 3-5-3]).
puz(5, [1-1, 15-15], 15-15, [9-1], [6-1], [9-3-5, 10-8-2]).
puz(6, [1-1, 10-10], 10-10, [9-6], [6-2], [6-1-5,
    10-5-4,7-9-4]).
puz(7, [1-1, 5-5], 5-5, [2-2,4-3], [], [1-3-3, 3-5-2]).
puz(8, [1-1, 3-3], 3-3, [], [1-2,2-1], [3-1-2, 1-3-2]).
puz(9,[1-1,9-9],9-9,[4-3,7-3],[2-3],[3-3-5,6-3-3,6-5-5]).

randomPuzzle:-random(1,8,Puzzle),
              bosnianSnake(Puzzle).

bosnianSnake(N) :-
puz(N, [BeginRow-BeginCol,EndRow-EndCol],NR-NC, RowCells,
    ColCells, CellsAround),
board(NR, NC, Board),
matrixToListOfLists(Board,List),
headAndTailCells(List, BeginRow,BeginCol,EndRow,EndCol,NR),
imposeConectivity(List,List,NR,1),
scrollCellsAround(CellsAround,List,NR),
scrollRestrictionsRow(RowCells,List,NR),
scrollRestrictionsCol(ColCells,List,NR),
count(1,List,#=,Count),
reset_timer,
labeling([minimize(Count)], List),
print_time,
fd_statistics,
list_to_matrix(List,NR,Board),
printFinalBoard(Board,1,1,CellsAround,RowCells,ColCells,NR).

reset_timer :- statistics(walltime,_).
print_time :-
        statistics(walltime,[_,T]),
        TS is ((T//10)*10),
        nl, write('Time: '), write(TS), write('ms'), nl, nl.

list_to_matrix([], _, []).
list_to_matrix(List, Size, [Row|Matrix]):-
```

```
43    list_to_matrix_row(List, Size, Row, Tail),
44    list_to_matrix(Tail, Size, Matrix).
45
46  list_to_matrix_row(Tail, 0, [], Tail).
47  list_to_matrix_row([Item|List], Size, [Item|Row], Tail):-
48    NSize is Size-1,
49    list_to_matrix_row(List, NSize, Row, Tail).
50
51  headAndTailCells(List, BeginRow,BeginCol,EndRow,EndCol,NR) :-
52  getPosition(NR,BeginRow,BeginCol,Position),
53  getPosition(NR,EndRow,EndCol,EndPosition),
54  nth1(Position,List,Element),
55  nth1(EndPosition,List,Element2),
56  Element=1,
57  Element2=1.
58
59  getRowAux(_,L,L,Size,Size).
60  getRowAux(List,ListaAux,ListOut,_,FinalRow):-
61    nth1(FinalRow,List,Element),
62    append([Element],ListaAux,Return),
63    FinalRow2 is FinalRow-1,
64    getRowAux(List,Return,ListOut,_,FinalRow2).
65
66  getRow(List,Row,ListOut,Size):-
67    Final is Row*Size,
68    First is Final-Size,
69    getRowAux(List,[],ListOut,First,Final).
70
71
72  getColAux(List,ListaAux,ListOut,First,FinalCol,Size):-
73    nth1(FinalCol,List,Element),
74    append([Element],ListaAux,Return),
75    FinalCol\=First,
76    FinalCol2 is FinalCol-Size,
77    getColAux(List,Return,ListOut,First,FinalCol2,Size).
78
79  getColAux(List,ListaAux,ListOut,First,FinalCol,_):-
80    nth1(FinalCol,List,Element),
81    append([Element],ListaAux,ListOut),
82    FinalCol==First.
83
84
85  getCol(List,Col,ListOut,Size):-
86    Dim is Size*Size,
87    Value is Dim-Size,
88    Final is Value+Col,
89    getColAux(List,[],ListOut,Col,Final,Size).
90
91  cellsOfRestrictionOut_ROW(List,Number,Row,Size) :-
92    Number2 is (Size - Number),
```

```
93    getRow(List,Row,ListOut,Size),
94    global_cardinality(ListOut,[1-Number, 0-Number2]).
95
96  cellsOfRestrictionOut_COL(List,Number,Col,Size) :-
97    Number2 is (Size - Number),
98    getCol(List,Col,ListOut,Size),
99    global_cardinality(ListOut,[1-Number, 0-Number2]).
100
101 scrollRestrictionsRow([],_,_).
102 scrollRestrictionsRow([Row-Number|Tail],List,Size):-
103   cellsOfRestrictionOut_ROW(List,Number,Row,Size),
104   scrollRestrictionsRow(Tail,List,Size).
105
106 scrollRestrictionsCol([],_,_).
107 scrollRestrictionsCol([Col-Number|Tail],List,Size):-
108   cellsOfRestrictionOut_COL(List,Number,Col,Size),
109   scrollRestrictionsCol(Tail,List,Size).
110
111 scrollCellsAround([],_,_).
112 scrollCellsAround([Row-Col-Number|Tail],List,Size):-
113   cellsAround(List,Row,Col,Number,Size),
114   scrollCellsAround(Tail,List,Size).
115
116 cellsAround(List, Nrow, Ncol, Number, NR) :-
117   getPosition(NR,Nrow,Ncol,Position),
118   Mod is Position mod NR,
119   Mod \=0,
120   Mod \=1,
121   Dim is NR*NR,
122   Value is Dim-NR,
123   Value1 is Value+1,
124   Position >NR,
125   Position < Value1,
126   Number2 is (8 - Number),
127   getAllNeighbours(Position, ListOut, List, NR),
128   global_cardinality(ListOut,[1-Number,0-Number2]).
129
130   cellsAround(List, Nrow, Ncol, Number, NR) :-
131     getPosition(NR,Nrow,Ncol,Position),
132     Dim is NR*NR,
133     Value is Dim-NR,
134     Value1 is Value+1,
135     Position\=1,
136     Position\=Value1,
137     Position\=NR,
138     Position\=Dim,
139     Mod is Position mod NR,
140     (Mod ==1;
141     Mod==0),
142     Number2 is (5 - Number),
```

```
143     getAllNeighbours(Position, ListOut, List, NR),
144     global_cardinality(ListOut,[1-Number,0-Number2]).
145
146
147 cellsAround(List, Nrow, Ncol, Number, NR) :-
148   getPosition(NR,Nrow,Ncol,Position),
149   Dim is NR*NR,
150   Value is Dim-NR,
151   Value1 is Value+1,
152   (Position==1;
153   Position==Value1;
154   Position==NR;
155   Position==Dim),
156   Number2 is (3 - Number),
157   getAllNeighbours(Position, ListOut, List, NR),
158   global_cardinality(ListOut,[1-Number,0-Number2]).
159
160
161 cellsAround(List, Nrow, Ncol, Number, NR) :-
162   getPosition(NR,Nrow,Ncol,Position),
163   Dim is NR*NR,
164   Value is Dim-NR,
165   Value1 is Value+1,
166   ((Position < NR,
167   Position\=1);
168   (Position > Value1,
169   Position\=Dim)),
170   Number2 is (5 - Number),
171   getAllNeighbours(Position, ListOut, List, NR),
172   nth1(Position,List,Element),
173   Element#=0,
174   global_cardinality(ListOut,[1-Number,0-Number2]).
175
176 board(_,0,[]).
177 board(Size, NumberOfLists, [HList|TList]) :-
178 length(HList, Size),
179 domain(HList, 0, 1),
180 TempNumberOfLists is NumberOfLists-1,
181 board(Size, TempNumberOfLists, TList).
```

Ficheiro "conectivity.pl"

```
1 :- use_module(library(clpfd)).
2 :- use_module(library(lists)).
3
4
5 matrixToListOfLists(Board,List) :-
```

```
append(Board,List).

getPosition(Dim,Row,Col,Position) :-
Position is (Row-1)*Dim+Col.

imposeConectivity([],_,_,_).

imposeConectivity([_|Tail],List,Dim,Position) :-
getNeighbours(Position,Neighbours,List,Dim),
getAllNeighbours(Position,AllNeighbours,List,Dim),
imposePosition(Position,Neighbours,AllNeighbours,List,Dim),
NextPosition is Position+1,
imposeConectivity(Tail,List,Dim,NextPosition).

imposePosition(Position,Neighbours,AllNeighbours,List,Size):-
  Dim is Size*Size,
  Value is Dim-Size,
  Value1 is Value +1,
  Position \=1,
  Position \=Size,
  Position \=Dim,
  Position\=Value1,
  sum(Neighbours,#=,Sum),
  sum(AllNeighbours,#=,Sum2),
  nth1(Position,List,Element),
  Element#=1 #=> ((Sum #=2) #/\ (Sum2 #=< 4)).

imposePosition(Position,Neighbours,AllNeighbours,List,Size):-
  Dim is Size*Size,
  Value is Dim-Size,
  Value1 is Value +1,
  (Position==Size;
  Position==Value1),
  sum(Neighbours,#=,Sum),
  sum(AllNeighbours,#=,Sum2),
  nth1(Position,List,Element),
  Element#=1 #=> ((Sum #=2) #/\ (Sum2 #= 2)).

imposePosition(Position,Neighbours,AllNeighbours,List,Size):-
Dim is Size*Size,
(Position==1;
Position==Dim),
sum(Neighbours,#=,Sum),
sum(AllNeighbours,#=,Sum2),
nth1(Position,List,Element),
Element#=1 #=> ((Sum #=1) #/\ (Sum2 #=< 2)).


getNeighbours(Position,Neighbours,List,Size):-
Position==1,
```

```
56  PositionRight is Position+1,
57  PositionDown is Position+Size,
58  setof(Element, (nth1(PositionRight,List,Element);
59                  nth1(PositionDown,List,Element)),
60                  Neighbours).
61
62  getNeighbours(Position,Neighbours,List,Size):-
63  Dim is Size*Size,
64  Position==Dim,
65  PositionLeft is Position-1,
66  PositionUp is Position-Size,
67  setof(Element, (nth1(PositionUp,List,Element);
68                  nth1(PositionLeft,List,Element)),
69                  Neighbours).
70
71  getNeighbours(Position,Neighbours,List,Size):-
72  Dim is Size*Size,
73  Value is Dim-Size,
74  Value1 is Value+1,
75  Position==Value1,
76  PositionRight is Position+1,
77  PositionUp is Position-Size,
78  setof(Element, (nth1(PositionUp,List,Element);
79                  nth1(PositionRight,List,Element)),
80                  Neighbours).
81
82  getNeighbours(Position,Neighbours,List,Size):-
83  Position==Size,
84  PositionLeft is Position-1,
85  PositionDown is Position+Size,
86  setof(Element, (nth1(PositionLeft,List,Element);
87                  nth1(PositionDown,List,Element)),
88                  Neighbours).
89
90  getNeighbours(Position,Neighbours,List,Size):-
91  Dim is Size*Size,
92  Value is Dim-Size,
93  Value1 is Value+1,
94  Mod is Position mod Size,
95  Mod ==1,
96  Position\=1,
97  Position\=Value1,
98  PositionRight is Position+1,
99  PositionDown is Position+Size,
100 PositionUp is Position-Size,
101 setof(Element, (nth1(PositionRight,List,Element);
102                 nth1(PositionDown,List,Element);
103                 nth1(PositionUp,List,Element)),
104                 Neighbours).
105
```

```
106 getNeighbours(Position,Neighbours,List,Size):-
107 Dim is Size*Size,
108 Mod is Position mod Size,
109 Mod ==0,
110 Position\=Size,
111 Position\=Dim,
112 PositionLeft is Position-1,
113 PositionDown is Position+Size,
114 PositionUp is Position-Size,
115 setof(Element, (nth1(PositionLeft,List,Element);
116                 nth1(PositionDown,List,Element);
117                 nth1(PositionUp,List,Element)),
118                 Neighbours).
119
120 getNeighbours(Position,Neighbours,List,Size):-
121 Position<Size,
122 Position\=1,
123 PositionLeft is Position-1,
124 PositionDown is Position+Size,
125 PositionRight is Position+1,
126 setof(Element, (nth1(PositionLeft,List,Element);
127                 nth1(PositionDown,List,Element);
128                 nth1(PositionRight,List,Element)),
129                 Neighbours).
130
131 getNeighbours(Position,Neighbours,List,Size):-
132 Dim is Size*Size,
133 Value is Dim-Size,
134 Value1 is Value+1,
135 Position>Value1,
136 Position\=Dim,
137 PositionLeft is Position-1,
138 PositionUp is Position-Size,
139 PositionRight is Position+1,
140 setof(Element, (nth1(PositionLeft,List,Element);
141                 nth1(PositionUp,List,Element);
142                 nth1(PositionRight,List,Element)),
143                 Neighbours).
144
145 getNeighbours(Position,Neighbours,List,Size):-
146 Position>Size,
147 Dim is Size*Size,
148 Value is Dim-Size,
149 Value1 is Value+1,
150 Position<Value1,
151 Mod is Position mod Size,
152 Mod \=0,
153 Mod \=1,
154 PositionLeft is Position-1,
155 PositionUp is Position-Size,
```

```
156 PositionDown is Position+Size,
157 PositionRight is Position+1,
158 setof(Element, (nth1(PositionLeft,List,Element);
159                  nth1(PositionUp,List,Element);
160                  nth1(PositionDown,List,Element);
161                  nth1(PositionRight,List,Element)),
162                  Neighbours).
163
164
165               %%%%%%%%
166
167 getAllNeighbours(Position,Neighbours,List,Size):-
168 Position==1,
169 PositionRight is Position+1,
170 PositionDown is Position+Size,
171 PositionDownRight is PositionRight+Size,
172 setof(Element, (nth1(PositionRight,List,Element);
173                  nth1(PositionDownRight,List,Element);
174                  nth1(PositionDown,List,Element)),
175                  Neighbours).
176
177 getAllNeighbours(Position,Neighbours,List,Size):-
178 Dim is Size*Size,
179 Position==Dim,
180 PositionLeft is Position-1,
181 PositionUp is Position-Size,
182 PositionUpLeft is PositionLeft-Size,
183 setof(Element, (nth1(PositionUp,List,Element);
184                  nth1(PositionUpLeft,List,Element);
185                  nth1(PositionLeft,List,Element)),
186                  Neighbours).
187
188 getAllNeighbours(Position,Neighbours,List,Size):-
189 Dim is Size*Size,
190 Value is Dim-Size,
191 Value1 is Value+1,
192 Position==Value1,
193 PositionRight is Position+1,
194 PositionUp is Position-Size,
195 PositionUpRight is PositionRight-Size,
196 setof(Element, (nth1(PositionUp,List,Element);
197                  nth1(PositionUpRight,List,Element);
198                  nth1(PositionRight,List,Element)),
199                  Neighbours).
200
201 getAllNeighbours(Position,Neighbours,List,Size):-
202 Position==Size,
203 PositionLeft is Position-1,
204 PositionDown is Position+Size,
205 PositionDownLeft is PositionDown-1,
```

```
206 setof(Element, (nth1(PositionLeft,List,Element);
207                 nth1(PositionDownLeft,List,Element);
208                 nth1(PositionDown,List,Element)),
209                 Neighbours).
210
211 getAllNeighbours(Position,Neighbours,List,Size):-
212 Dim is Size*Size,
213 Value is Dim-Size,
214 Value1 is Value+1,
215 Mod is Position mod Size,
216 Mod ==1,
217 Position\=1,
218 Position\=Value1,
219 PositionRight is Position+1,
220 PositionDown is Position+Size,
221 PositionUp is Position-Size,
222 PositionRightUp is PositionUp+1,
223 PositionDownRight is PositionDown+1,
224 setof(Element, (nth1(PositionRight,List,Element);
225                 nth1(PositionRightUp,List,Element);
226                 nth1(PositionDownRight,List,Element);
227                 nth1(PositionDown,List,Element);
228                 nth1(PositionUp,List,Element)),
229                 Neighbours).
230
231 getAllNeighbours(Position,Neighbours,List,Size):-
232 Dim is Size*Size,
233 Mod is Position mod Size,
234 Mod ==0,
235 Position\=Size,
236 Position\=Dim,
237 PositionLeft is Position-1,
238 PositionDown is Position+Size,
239 PositionUp is Position-Size,
240 PositionUpLeft is PositionUp-1,
241 PositionDownLeft is PositionDown-1,
242 setof(Element, (nth1(PositionLeft,List,Element);
243                 nth1(PositionUpLeft,List,Element);
244                 nth1(PositionDownLeft,List,Element);
245                 nth1(PositionDown,List,Element);
246                 nth1(PositionUp,List,Element)),
247                 Neighbours).
248
249 getAllNeighbours(Position,Neighbours,List,Size):-
250 Position<Size,
251 Position\=1,
252 PositionLeft is Position-1,
253 PositionDown is Position+Size,
254 PositionRight is Position+1,
255 PositionDownLeft is PositionDown-1,
```

```
256 PositionDownRight is PositionDown+1,
257 setof(Element, (nth1(PositionLeft,List,Element);
258                 nth1(PositionDownLeft,List,Element);
259                 nth1(PositionDownRight,List,Element);
260                 nth1(PositionDown,List,Element);
261                 nth1(PositionRight,List,Element)),
262                 Neighbours).
263
264 getAllNeighbours(Position,Neighbours,List,Size):-
265 Dim is Size*Size,
266 Value is Dim-Size,
267 Value1 is Value+1,
268 Position>Value1,
269 Position\=Dim,
270 PositionLeft is Position-1,
271 PositionUp is Position-Size,
272 PositionRight is Position+1,
273 PositionUpRight is PositionUp+1,
274 PositionUpLeft is PositionUp-1,
275 setof(Element, (nth1(PositionLeft,List,Element);
276                 nth1(PositionUpLeft,List,Element);
277                 nth1(PositionUpRight,List,Element);
278                 nth1(PositionUp,List,Element);
279                 nth1(PositionRight,List,Element)),
280                 Neighbours).
281
282 getAllNeighbours(Position,Neighbours,List,Size):-
283 Position>Size,
284 Dim is Size*Size,
285 Value is Dim-Size,
286 Value1 is Value+1,
287 Position<Value1,
288 Mod is Position mod Size,
289 Mod \=0,
290 Mod \=1,
291 PositionLeft is Position-1,
292 PositionUp is Position-Size,
293 PositionDown is Position+Size,
294 PositionRight is Position+1,
295 PositionDownRight is PositionDown+1,
296 PositionDownLeft is PositionDown-1,
297 PositionUpRight is PositionUp+1,
298 PositionUpLeft is PositionUp-1,
299 setof(Element, (nth1(PositionLeft,List,Element);
300                 nth1(PositionUp,List,Element);
301                 nth1(PositionDown,List,Element);
302                 nth1(PositionRight,List,Element);
303                 nth1(PositionUpLeft,List,Element);
304                 nth1(PositionUpRight,List,Element);
305                 nth1(PositionDownLeft,List,Element);
```

```
306                     nth1(PositionDownRight,List,Element)),
307                     Neighbours).
```

Ficheiro "board.pl"

```
1  printFinalBoard([L|Ls],Row,Col,CellsAround,RowCells,ColCells,
       Size):-
2    nl,
3    write('     '),
4    printCol(ColCells,Size,1),nl,
5    printBoard([L|Ls],Row,Col,CellsAround,RowCells,Size),
6    printLine(Size).
7
8
9  printBoard([],_,_,_,_,_).
10 printBoard([L|Ls],Row,Col,CellsAround,[Row1-Number|Tail],Size
       ):-
11       Row==Row1,
12       printLine(Size),nl,
13       write('    |'),
14       printSpaces(L),nl,
15       write('  '),
16       write(Number),
17       write(' |'),
18       printRow(L,Row,Col,CellsAround,CellsAround2),nl,
19       write('    |'),
20       printSpaces(L),nl,
21       Row2 is Row+1,
22       printBoard(Ls,Row2,Col,CellsAround2,Tail,Size).
23
24 printBoard([L|Ls],Row,Col,CellsAround,[Row1-Number|Tail],Size
       ):-
25       Row\=Row1,
26       printLine(Size),nl,
27       write('    |'),
28       printSpaces(L),nl,
29       write('    |'),
30       printRow(L,Row,Col,CellsAround,CellsAround2),nl,
31       write('    |'),
32       printSpaces(L),nl,
33       Row2 is Row+1,
34       printBoard(Ls,Row2,Col,CellsAround2,[Row1-Number|Tail],
             Size).
35
36 printBoard([L|Ls],Row,Col,CellsAround,[],Size):-
37       printLine(Size),nl,
38       write('    |'),
```

```
39          printSpaces(L),nl,
40          write('    |'),
41          printRow(L,Row,Col,CellsAround,CellsAround2),nl,
42          write('    |'),
43          printSpaces(L),nl,
44          Row2 is Row+1,
45          printBoard(Ls,Row2,Col,CellsAround2,[],Size).
46
47  printCol(_,S,S).
48  printCol([Col-Number|Tail],Size,N):-
49          N==Col,
50          write('  '),
51          write(Number),
52          write('  '),
53          N1 is N+1,
54          printCol(Tail,Size,N1).
55
56  printCol([Col-Number|Tail],Size,N):-
57          N\=Col,
58          write('     '),
59          N1 is N+1,
60          printCol([Col-Number|Tail],Size,N1).
61
62  printCol([],Size,N):-
63          write('     '),
64          N1 is N+1,
65          printCol([],Size,N1).
66
67  printRow([],_,_,S,S).
68  printRow([_|Xs],Row,Col,[Row1-Col1-Number|Tail],S):-
69          Row==Row1,
70          Col==Col1,
71          write('  '),
72          write(Number),
73          write(' |'),
74          Col2 is Col+1,
75          printRow(Xs,Row,Col2,Tail,S).
76
77  printRow([X|Xs],Row,Col,[],S):-
78          X==0,
79          write('    '),write('|'),
80          Col2 is Col+1,
81          printRow(Xs,Row,Col2,[],S).
82
83  printRow([X|Xs],Row,Col,[],S):-
84          X==1,
85          write('****'),write('|'),
86          Col2 is Col+1,
87          printRow(Xs,Row,Col2,[],S).
88
```

```
89  printRow([X|Xs],Row,Col,[Row1-Col1-Number|Tail],S):-
90          ((Row\=Row1,
91          Col\=Col1);
92          (Row==Row1,
93          Col\=Col1);
94          (Row\=Row1,
95          Col==Col1)),
96          X==0,
97          write('    '),write('|'),
98          Col2 is Col+1,
99          printRow(Xs,Row,Col2,[Row1-Col1-Number|Tail],S).
100
101 printRow([X|Xs],Row,Col,[Row1-Col1-Number|Tail],S):-
102       ((Row\=Row1,
103       Col\=Col1);
104       (Row==Row1,
105       Col\=Col1);
106       (Row\=Row1,
107       Col==Col1)),
108       X==1,
109       write('****'),write('|'),
110       Col2 is Col+1,
111       printRow(Xs,Row,Col2,[Row1-Col1-Number|Tail],S).
112
113
114 printSpaces([]).
115 printSpaces([X|Xs]):-
116 X==0,
117 write('    |'),
118 printSpaces(Xs).
119
120 printSpaces([X|Xs]):-
121 X==1,
122 write('****|'),
123 printSpaces(Xs).
124
125 printLine(Size):-
126   write('    '),
127   printLineAux(Size).
128 printLineAux(0).
129 printLineAux(Size):-
130   write('-----'),
131   S is Size-1,
132   printLineAux(S).
```