Resolução de Problema de Decisão usando Programação em Lógica com Restrições
Bosnian Road

Anabela Costa e Silva, e Beatriz Souto de Sá Baldaia

Faculdade de Engenharia da Universidade do Porto, R. Dr. Roberto Frias, 4200-465 Porto, Portugal,
candidato@fe.up.pt,
WWW home page: https://sigarra.up.pt/feup/pt/web_page.Inicial,
FEUP-PLOG, Turma 3MIEIC4, Grupo BosnianRoad_1

Resumo O artigo aborda a construção de um programa em Programação em Lógica com Restrições (PLR) para a resolução do problema/puzzle de decisão combinatória Bosnian Road, no âmbito da cadeira Programação em Lógica. Este puzzle baseia-se em traçar um caminho único e fechado, num tabuleiro quadrado, que segue restrições espaciais que serão descritas à frente. Para tal recorremos ao mecanismo "constrain and generate" de PLR a partir da instância *Solver* clp(FD) do esquema geral de PLR (CLP), disponível como uma biblioteca, clpfd. No entanto, devido à extrema complexidade do problema, apresentamos uma segunda solução que também usa o mecanismo de unificação do Prolog, com pesquisa do tipo "generate and test".
O trabalho não foi finalizado com completo sucesso pois há casos em que se torna impossivel, a partir de restrições, detetar a existência de um só caminho fechado. Assim, para estes casos, é necessário testar o resultado após os valores do mesmo estarem instânciados.

Keywords: prolog, cplfd, restrições

1 Introdução

O objetivo deste trabalho foi a resolução do puzzle Bosnian Road em Programação em Lógica com Restrições (PLR), através do mecanismo "constrain and generate", recorrendo aos predicados da biblioteca cplfd para testar a consistência e o vínculo de restrições sobre domínios finitos, bem como para obter soluções atribuindo valores às variáveis do problema.
A motivação deste projeto foi explorar a programação em lógica com restrições. Esta classe de linguagens de programação combina a declaratividade da programação em lógica com a eficiência da resolução de restrição, sendo por isso usual na resolução de problemas NP-completos.
Bosnian Road consiste no desenho de um único caminho fechado que conect células horizontalmente e verticalmente. O caminho não se pode tocar a ele próprio/intersetar, nem diagonalmente. Também não pode transpôr as "clue

cells".
O artigo visa explicar a resolução do trabalho e apresentar os resultados alcançados, seguindo a seguinte estrutura:

- **Introdução -** Descrição dos objetivos e motivação do trabalho, referência ao problema em análise e descrição da estrutura do resto do artigo.
- **Descrição do Problema -** Descrição detalhada do problema de decisão em análise.
- **Abordagem -** Descrição da modelação do problema como um Problema de Satisfação de Restrições (PSR), abordando os tópicos: variáveis de decisão e os seus domínios; implementação de restrições rígidas e flexíveis, utilizando o SICStus Prolog; avaliação da solução obtida; estratégia de etiquetagem (labeling).
- **Visualização da Solução -** Explicação dos predicados que permitem visualizar a solução em modo de texto.
- **Resultados -** Exemplos de aplicação em instâncias do problema com diferentes complexidades e resultados obtidos.
- **Conclusões e Trabalho Futuro -** Conclusões retiradas dos resultados obtidos e possíveis melhorias.

2 Descrição do Problema

O objetivo do puzzle Bosnian Road é traçar um único caminho fechado, num tabuleiro quadrado em que cada célula representa uma unidade. Este caminho tem de conectar células do tabuleiro, vertical e horizontalmente, e não se pode tocar em/intersetar com ele próprio, nem mesmo diagonalmente.
O caminho também não pode transpôr "clue cells", isto é, células numeradas cujo número representa o número de casas vizinhas que pertencem ao caminho. Assim, estas células podem assumir valores de 1 a 8.
Também existem células inicialmente preenchidas que funcionam como obstáculos, ou seja, o caminho não os pode transpôr.

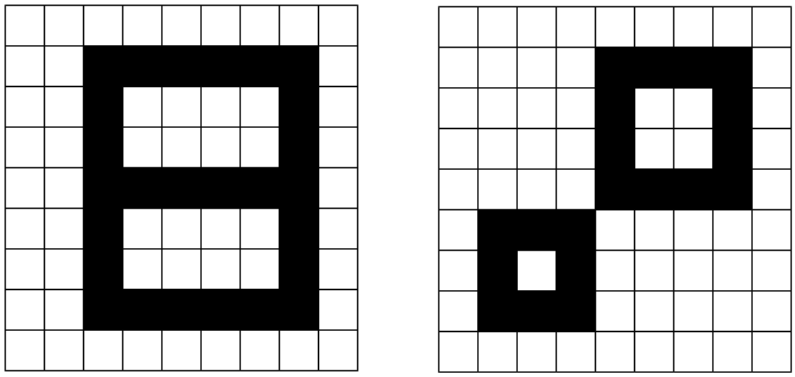

Figura 1. Restrição de interseções impossiveis.

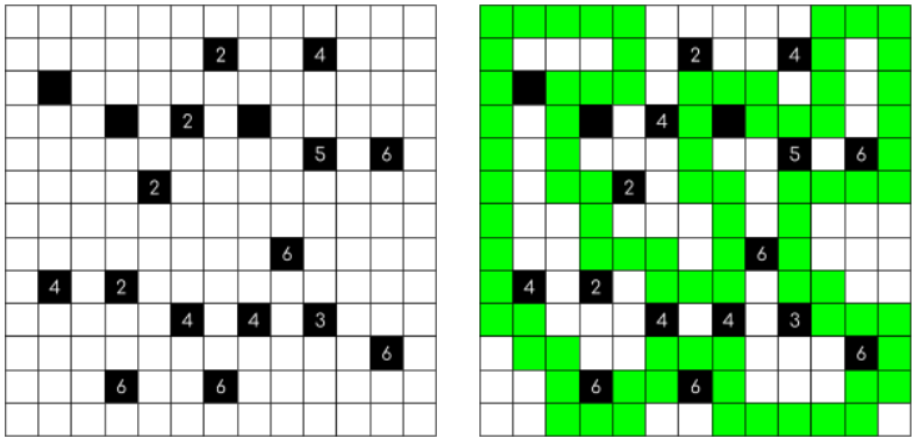

Figura 2. Caminho traçado seguindo restrições impostas por células especiais.

3 Abordagem

O problema em questão trata-se de um Problema de Satisfação de Restrições (PSR) pois pertence a classe cujo solver clp(FD) é o mais adquado para a resolução. Nesta abordagem são escolhidos valores, de domínios pré-definidos, para certas variáveis de forma a que as restrições sobre as variáveis sejam todas satisfeitas. Deste modo, é necessário declarar as variáveis e seus domínios, colocar as restrições do problema e pesquisar uma solução possível através de pesquisa com retrocesso ou uma solução ótima usando pesquisa tipo *branch-and-bound*.
Na implementação em PROLOG representamos o tabuleiro como uma lista de

listas. Para facilitar a implementação e uso de certos predicados, também usamos uma lista que traduz a concatenação das listas da matriz anterior. Cada elemento pode assumir os valores 0 (célula que não pertence ao caminho) ou 1 (célula do caminho). Para representar as pista usamos uma lista de pistas onde cada pista é representada por um elemento composto do tipo X-Y-V, em que X representa a coluna, Y a linha e V o valor da pista. Usamos também uma lista para representar as casa fechadas, sendo cada uma delas um elemento do tipo X-Y.

3.1 Variáveis de Decisão

Num tabuleiro de largura N, a solução contém N x N variáveis de decisão (uma para cada casa do tabuleiro). Elas tem um domínio entre 0 (casa vazia) e 1 (pertence ao caminho).

3.2 Restrições

Organizamos as restrições impostas ao tabuleiro em 4 predicados:

- **checkNumberedPositions(Clues,Board,Dim)**
  Percorremos as "clue cells"e obrigamos que o número de células vizinhas a estas e que simultaneamente pertencem ao caminho seja igual ao número atribuido a estas células especiais.

```
Elem #= 0,
V #= ValueUpLeft + ValueUp + ValueUpRight +
ValueLeft + ValueRight + ValueDownLeft +
ValueDown + ValueDownRight.
```

- **checkZeroPositions(Zeros,Board,Dim)**
  Restringimos a solução para que as casa fechadas (casa inicialmente preenchidas a preto) não possam pertencer ao caminho.

```
Pos #= (Y-1) * Dim + X,
element(Pos, QueueBoard, 0)
```

- **checkIntersectedRoads(Board, Dim)**
  Impomos que caminho seja fechado e que não se toque, nem mesmo diagonalmente. Sabemos que o caminho é fechado se cada casa pertencente ao caminho tem duas casas, horizontalmente ou verticalmente, adjacentes também pertencentes ao caminho. Garantimos que o caminho não se interseta se para qualquer área 2 x 2, caso haja duas células diagonais pretas (valor 1), obrigatóriamente uma, e só uma, das duas restantes terá de ser preta (pertencer ao caminho).

```
VSides #= ValueLeft + ValueRight + ValueUp + ValueDown,
VDiagonal #= ValueUpLeft + ValueUpRight +
ValueDownLeft + ValueDownRight,
((Elem #= 1 #/\ VSides #= 2)#\/ (Elem #= 0)),
((Elem #= 1 #/\ ValueDownRight #= 1) #=>
(ValueRight #= 1 #\ ValueDown #= 1)),
((Elem #= 0 #/\ ValueDownRight #= 0) #=>
(ValueRight #= 0 #\/ ValueDown #= 0))
```

– **checkSimpleSeparatedRoads(Board)**
Contamos o número de transições de uma coluna/linha "limpa", isto é, sem nenhuma casa a preto (pertencente ao caminho), para uma coluna/linha com pelo menos uma casa preta. De seguida obrigamos que essa transição seja igual a 1, o que significa que só encontramos o inicio de um só caminho.
No entanto, esta restrição não resolve os casos de um caminho contido ou parcialmente rodeado por outro:

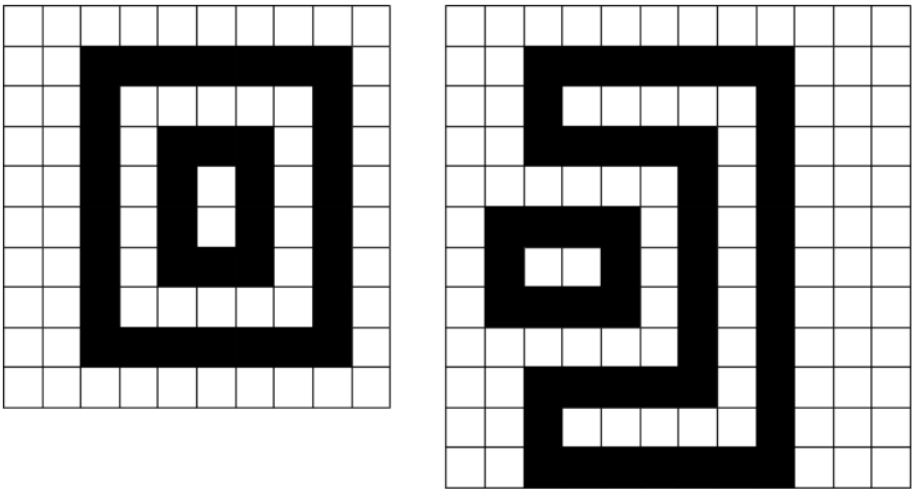

Figura 3. Exemplo de caminhos independentes indetetáveis.

3.3 Função de Avaliação

Devido à elevada complexidade, não conseguimos criar restrições que impedissem a existêmcia de certos caminhos independentes no mesmo tabuleiro, como os referidos anteriormente na subsecção 3.2. Pensamos em formas de atacar o problema, como o uso do *sorting/3* e do *circuit/2*, mas para ambos precisariamos de saber previamente o número de casas de cada caminho separadamente.

Também pensamos recorrer ao algoritmo *flood fill*, mas requeriria o uso de variáveis aulxiliares que ocupariam muita memória e tornariam o código ineficiente e moroso.
Ainda surgiu a ideia de uso de automatos para o salto de uma casa do caminha para outra, também do caminho, adjacente a essa. Na teoria deveria resultar, mas a implementação revelar-se-ia deveras complexa e também não muito viável devido aos recursos necessário.
Assim, optamos por restringir ao máximo o problema, de forma a reduzir o espaço de pesquisa, e após etiquetagem (labeling) percorrer o tabuleiro resultante e guardar numa lista as posições das células a preto (**checkResultValues(Board, Dim, It)**), começar na primeira casa guardada nessa lista e saltar para uma sua adjacente que também se encontrasse na lista e que ainda não tivesse sido visitada. Tal pesquisa terminaria quando não houvessem mais casas para onde pudessemos transitar (fim de um caminho fechado). Os indíces das casas visitadas vão sendo guardados também numa lista. Por fim, o predicado **uniqueRoad/2** falharia se o número de casas visitadas nesta pesquisa fosse diferente do número de casas guardadas.

3.4 Estratégia de Pesquisa

Testamos várias estratégias de pesquisa, usando múltiplas opções do predicado labeling. Como é possível verificar na tabela 3, todas as opções devolvem valores próximos.

4 Visualização da Solução

Os predicados que implementam a visualização do resultado encontram-se no ficheiro print.pl(A.4).
Como separamos a solução das pistas e casas fechadas, antes de mostrar a solução criamos um tabuleiro auxiliares que contém as pistas, casas fechadas e a solução. O predicado **createPrintBoard(Solution, Board-Zeros, Dim, Square)** devolve em Square um tabuleiro contendo as variáveis responsáveis para a mostra do tabuleiro.
Na figura 10 vê-se um exemplo de visualização de uma solução.

Valor	Significado	Representação
1 a 8	Casa com pista	Número de 1 a 8
9	Casa fechada	■
10	Casa vazia	Espaço
11	Casa do caminho	▩

Tabela 1. Valores das casas no tabuleiro a mostrar

5 Resultados

Foram efectuados testes para avaliar a complexidade da nossa resolução para este problema de decisão.

Tabela 2. Complexidade da resolução dos puzzles

Dimensão	Tempo	Reatamentos	Implicações	Podas	Retrocessos	Restrições Criadas
25	7.14	933368	704656	668255	1386	75493
24	0.63	2653064	2174640	1901775	3819	86377
23	0.51	55775	30360	40947	10	22978
20	0.24	45324	24774	33030	13	17799
15	0.04	23513	12211	17081	3	10084
10	0.00	9002	4712	6489	0	4168
5	0.00	2576	1242	1751	0	1094

As figuras 4, 5, 6, 7, 8, e 9 são os gráficos onde se mostra a progressão das métricas recolhidas.

6 Conclusões e Trabalho Futuro

Programação em Lógica com Restrições é marcada pelo reduzido tempo de desenvolvimento, facilidade de manutenção, eficiência na resolução, clareza e brevidade dos programas tal como podemos verificar ao longo do desenvolvimento do projeto. Apesar de todos os beneficios de um problema de satisfação de restrições, a solução do mesmo é encontrada através de uma pesquisa sistemática, usualmente guiada por uma heurística, a partir de todas as atribuições de valores às variáveis. Deste modo, todos os valores atribuidos às variáveis de domínio devem satisfazer as restrições, no entanto, mesmo após a computação determinística e pesquisa podemos enfrentar, como no nosso caso, falta de cumprimento de certos parâmetros do enunciado devido ao facto de as restrições não serem as suficientes.

Assim, concluimos que a eficiência da programação em lógica com restrições depende também da complexidade do problema na medida em que a tentativa de cobrir uma condição de extrema dificuldade leva-nos a um uso excessivo de recursos e a um código complicado, o que se torna menos viável do que recorrer a uma pesquisa posterior auxiliar do tipo *"generate and test"*.

A nossa solução proposta tem a vantagem de conseguir detetar as soluções erradas, mas ao mesmo tempo tem a limitação de percorrer todas estas, após o labeling, para inferir tal conclusão.

O trabalho desenvolvido poderia ser melhorado se fosse possível encontrar um método de restrinção de forma a impedir casos especiais de caminhos fechados e independentes, como um caminho contido no interior de outro, com o menor

gasto de memória e CPU possível, tentando não efectuar a análise repetitiva das mesmas células e manter um balanço entre recursos necessário e complexidade/-legibilidade do código.

Referências

1. Puzzle No. 438 : Bosnian Road.
   https://prasannaseshadri.wordpress.com/2013/09/15/puzzle-no-438-bosnian-road/
2. Bosnian Road.
   https://yureklis.wordpress.com/2013/01/21/bosnian-road/
3. Shading and Loops by Walker Anderson.
   http://logicmastersindia.com/lmitests/dl.asp?attachmentid=645&view=1
4. SWI-Prolog. 1987.
   http://matuszek.org/prolog/prolog-writing.html

A Anexos

A.1 Gráficos

Figura 4. Tempo em função da dimensão do tabuleiro

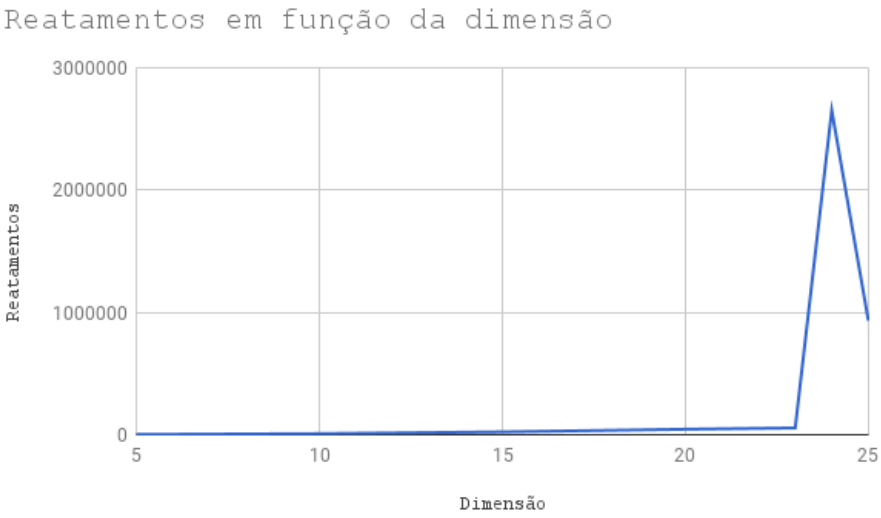

Figura 5. Reatamentos em função da dimensão do tabuleiro

Figura 6. Implicações em função da dimensão do tabuleiro

Figura 7. Podas em função da dimensão do tabuleiro

Figura 8. Retrocessos em função da dimensão do tabuleiro

Figura 9. Restrições criadas em função da dimensão

A.2 Imagens

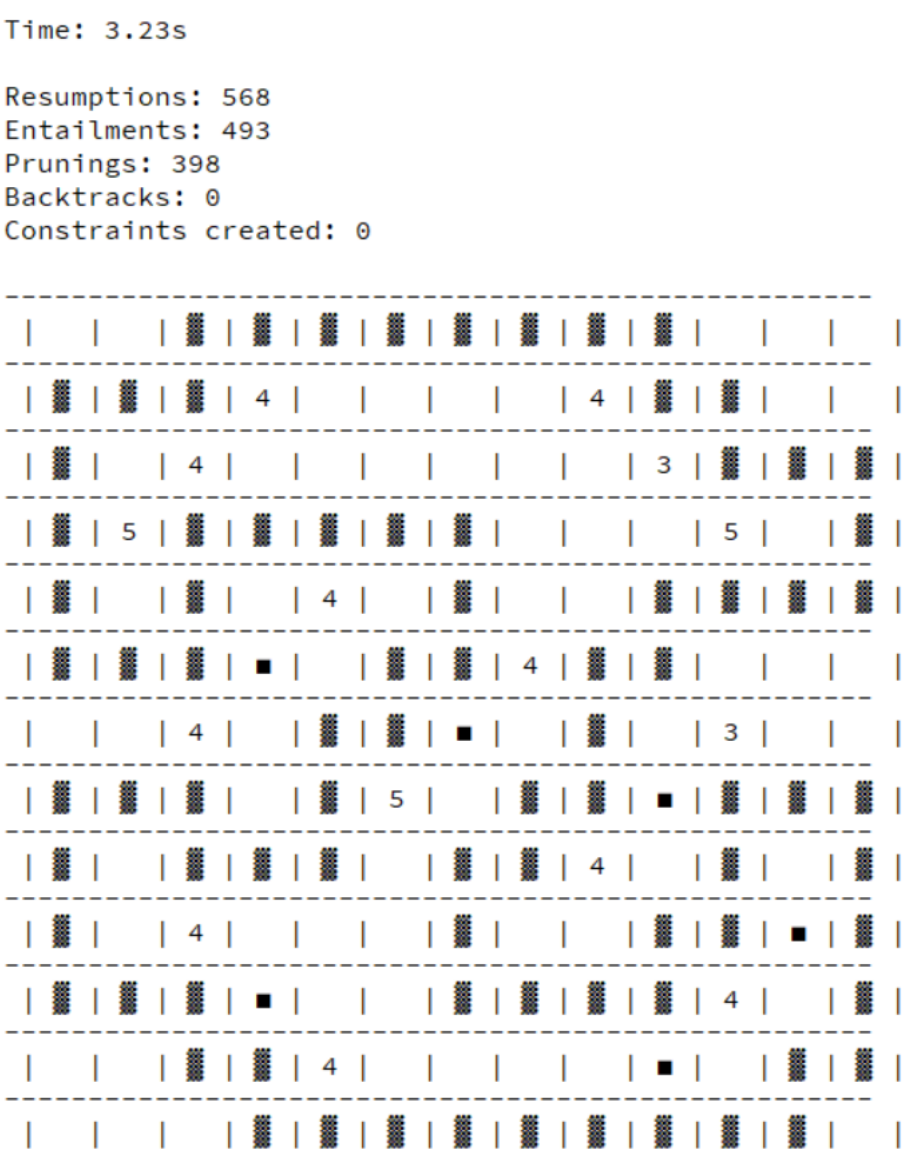

Figura 10. Exemplo da visualização de um puzzle

A.3 Tabelas

Tabela 3. Alteração das opções de labeling para a pesquisa

Time	leftmost	min	max	first_fail	anti_first_fail	occurrence	ffc	max_regret
step	”5,54”	”5,6”	”7,02”	”6,95”	”6,99”	”5,77”	”5,58”	”7,05”
enum	”7,17”	”5,69”	7,26	”5,65”	”5,68”	”5,74”	”5,66”	”5,66”
bisect	”5,48”	”5,59”	”7,08”	”5,54”	”5,76”	”6,9”	”7,1”	”5,51”
middle	”5,52”	”6,97”	”6,43”	”6,96”	”7,09”	”6,98”	”7,1”	”5,53”

Resumptions	leftmost	min	max	first_fail	anti_first_fail	occurrence	ffc	max_regret
step	296213	296213	296213	296213	296213	296213	296213	296213
enum	296213	296213	296213	296213	296213	296213	296213	296213
bisect	296213	296213	296213	296213	296213	296213	296213	296213
middle	296213	296213	296213	296213	296213	296213	296213	296213

Entailments	leftmost	min	max	first_fail	anti_first_fail	occurrence	ffc	max_regret
step	218457	218457	218457	218457	218457	218457	218457	218457
enum	218457	218457	218457	218457	218457	218457	218457	218457
bisect	218457	218457	218457	218457	218457	218457	218457	218457
middle	218457	218457	218457	218457	218457	218457	218457	218457

Prunings	leftmost	min	max	first_fail	anti_first_fail	occurrence	ffc	max_regret
step	214003	214003	214003	214003	214003	214003	214003	214003
enum	213596	213596	213596	213596	213596	213596	213596	213596
bisect	214525	214525	214525	214525	214525	214525	214525	214525
middle	214003	214003	214003	214003	214003	214003	214003	214003

Backtracks	leftmost	min	max	first_fail	anti_first_fail	occurrence	ffc	max_regret
step	407	407	407	407	407	407	407	407
enum	407	407	407	407	407	407	407	407
bisect	407	407	407	407	407	407	407	407
middle	407	407	407	407	407	407	407	407

Constraints created	leftmost	min	max	first_fail	anti_first_fail	occurrence	ffc	max_regret
step	28569	28569	28569	28569	28569	28569	28569	28569
enum	28569	28569	28569	28569	28569	28569	28569	28569
bisect	28569	28569	28569	28569	28569	28569	28569	28569
middle	28569	28569	28569	28569	28569	28569	28569	28569

A.4 Código

trace_road.pl Código que resolve um puzzle.

```
:- use_module(library(clpfd)).
:- use_module(library(lists)).

:-include('calculate_values.pl').
:-include('create_board.pl').
:-include('intersection_constraint.pl').
:-include('marked_positions_constraint.pl').
:-include('matrix_parser.pl').
:-include('numbered_positions_neighbors_constraint.pl').
:-include('print.pl').
:-include('simple_separated_roads_constraint.pl').
:-include('test_result.pl').
:-include('statistic.pl').

ite(If, Then, _):- If,!,Then.
ite(_,_,Else):- Else.

%funcao principal
calculate_road(Board-Zeros, Dim, Answer) :-
  length(Answer, Dim),
  createBoard(Answer, Dim),
  append(Answer, Res),%Res e uma lista com todos os elementos da matriz Answer
  checkNumberedPositions(Board,Res,Dim),
  checkZeroPositions(Zeros,Res,Dim),
  checkIntersectedRoads(Res, Dim),
  checkSimpleSeparatedRoads(Answer),
  reset_timer,
  labeling([leftmost,bisect], Res),
  print_time,
  fd_statistics.



%funcao principal com teste apos labeling
calculate_road_testing(Board-Zeros, Dim, Answer) :-
  length(Answer, Dim),
  createBoard(Answer, Dim),
  append(Answer, Res),%Res e uma lista com todos os elementos da matriz Answer
  checkNumberedPositions(Board,Res,Dim),
  checkZeroPositions(Zeros,Res,Dim),
  checkIntersectedRoads(Res, Dim),
  checkSimpleSeparatedRoads(Answer),
  reset_timer,
```

```
  labeling([], Res),
  uniqueRoad(Res, Dim),
  print_time,
  fd_statistics.


%['trace_road.pl'],calculate_road([2-2-5,5-2-5,3-3-3,2-5-3,6-6-2]-[],6,RES),
%showBoard(RES, [2-2-5,5-2-5,3-3-3,2-5-3,6-6-2]-[],6).
%['trace_road.pl'],calculate_road([2-2-5,5-2-5,3-3-3,2-5-3]-[],6,RES)
%,showBoard(RES, [2-2-5,5-2-5,3-3-3,2-5-3]-[],6).


/*['trace_road.pl'],calculate_road([2-4-5,3-3-4,3-7-4,3-10-4,4-2-4,5-5-4,
5-12-4,6-8-5,8-6-4,
9-2-4,9-9-4,10-3-3,11-4-5,11-7-3,11-11-4]-[4-6,4-11,7-7,10-8,
10-12,12-10],13,RES),
showBoard(RES,[2-4-5,3-3-4,3-7-4,3-10-4,4-2-4,5-5-4,5-12-4,6-8-5,8-6-4,
9-2-4,9-9-4,10-3-3,11-4-5,11-7-3,11-11-4]-[4-6,4-11,7-7,10-8,
10-12,12-10],13).*/
```

calculate_values.pl Código que devolve o valor das casas adjacentes a atual.

```
%obtem os valores das casas visinhas a casa de posicao (X, Y)
getBoardValues(X, Y, Res, Dim, Elem, Pos, ValueUpLeft, ValueUp,
ValueUpRight, ValueLeft, ValueRight, ValueDownLeft, ValueDown,
ValueDownRight) :-
  Pos #= (Y-1) * Dim + X,
  Left #= Pos - 1,
  Right #= Pos + 1,
  Up #= (Y-2) * Dim + X,
  UpLeft #= Up - 1,
  UpRight #= Up + 1,
  Down #= (Y) * Dim + X,
  DownLeft #= Down - 1,
  DownRight #= Down + 1,
  element(Pos, Res, Elem),

  ite(X = 1, ValueLeft = 0, element(Left, Res, ValueLeft)),

  ite(X = Dim, ValueRight = 0, element(Right, Res, ValueRight)),

  ite(Y = 1,
    (ValueUpLeft = 0, ValueUp = 0, ValueUpRight = 0),
  (element(Up, Res, ValueUp),
      ite(X = 1, ValueUpLeft = 0, element(UpLeft, Res, ValueUpLeft)),
      ite(X = Dim, ValueUpRight = 0, element(UpRight, Res, ValueUpRight)))),

  ite(Y = Dim,
    (ValueDownLeft = 0, ValueDown = 0, ValueDownRight = 0),
  (element(Down, Res, ValueDown),
     ite(X = 1, ValueDownLeft = 0, element(DownLeft, Res, ValueDownLeft)),
       ite(X = Dim, ValueDownRight = 0, element(DownRight, Res, ValueDownRight)))).
```

create_board.pl Código que gera puzzles BosnianRoad.

```
%['create_board.pl'],createBoard(4,B-_,A).
%['trace_road.pl'],createBoard(20,K,A).
%createBoard(9,K,A), calculate_road(K,9,RES),showBoard(RES, K,9).
:-use_module(library(random)).
createBoard(Dim, Board-[],Square):-
  length(Square, Dim),
  createBoard(Square, Dim),
  append(Square, AuxList),
  checkIntersectedRoads(AuxList, Dim),
  checkSimpleSeparatedRoads(Square),
  LowerBound is Dim // 2,
  UpperBound is 2 * Dim,
  repeat,
  random(LowerBound, UpperBound, NumberOfRoadBlocks),
  placeRoadBlocks(NumberOfRoadBlocks, AuxList),
  NumberOfClues is Dim * Dim // 7,
  placeClues(Board, NumberOfClues, AuxList, Dim),
  labeling([],AuxList).

placeRoadBlocks(0,_).
placeRoadBlocks(Number, Board):-
  random_member(1, Board),
  NextNumber is Number - 1,
  placeRoadBlocks(NextNumber, Board).

placeClues([], 0, _, _).
placeClues([X-Y-V|Old], Number, AuxList, BoardDimention):-
  AuxNumber is BoardDimention + 1,
  random(1, AuxNumber, X),
  random(1, AuxNumber, Y),
  getBoardValues(X, Y, AuxList, BoardDimention, Elem, Pos, ValueUpLeft, ValueUp, ValueUpRigh
  Elem #= 0,
  V #= ValueUpLeft + ValueUp + ValueUpRight + ValueLeft + ValueRight + ValueDownLeft + Valu
  V #> 0,
  NextNumber is Number - 1,
  placeClues(Old, NextNumber, AuxList, BoardDimention).
```

intersection_constraint.pl Código que restringe uma possível solução a ruas fechadas que não se intersectam.

```
%verifica se nao ha dois caminhos distintos com um vertice e comum
checkIntersectedRoads(Res, Dim):-
checkNextRow(Res, Dim, 1).

checkNextRow(_, Dim, Y):- Dim + 1 =:= Y.
checkNextRow(Res, Dim, Y):-
checkNextColumn(Res, Dim, Y, 1),
Y1 is Y + 1,
checkNextRow(Res, Dim, Y1).

checkNextColumn(_, Dim,_, X):- Dim + 1 =:= X. %chegou ao fim da linha
checkNextColumn(Res, Dim, Y, X):-
  getBoardValues(X, Y, Res, Dim, Elem, Pos, ValueUpLeft, ValueUp,
  ValueUpRight, ValueLeft, ValueRight, ValueDownLeft, ValueDown,
  ValueDownRight),

  VSides #= ValueLeft + ValueRight + ValueUp + ValueDown,
  VDiagonal #= ValueUpLeft + ValueUpRight + ValueDownLeft + ValueDownRight,
  ((Elem #= 1 #/\ VSides #= 2)#\/ (Elem #= 0)),
  ((Elem #= 1 #/\ ValueDownRight #= 1) #=> (ValueRight #= 1 #\ ValueDown #= 1)),
  ((Elem #= 0 #/\ ValueDownRight #= 0) #=> (ValueRight #= 0 #\/ ValueDown #= 0)),
  X1 is X + 1,
  checkNextColumn(Res, Dim, Y, X1).
```

marked_positions_constraint.pl Código que impõe a restrição representadas pelas casa fechadas/obestáculo.

```
%as posicoes que ja estao inicialmente marcadas a preto
%no mapa, nao podem fazer parte do caminho calculado
checkZeroPositions([], _, _).
checkZeroPositions([H|R], QueueBoard, Dim):-
  checkZero(H, QueueBoard, Dim),
  checkZeroPositions(R, QueueBoard, Dim).
checkZero(X-Y, QueueBoard, Dim):-
    Pos #= (Y-1) * Dim + X,%posicao correspondente na QueueBoard
    element(Pos, QueueBoard, 0).
```

matrix_parser.pl Código que cria o tabuleiro.

```
% Cria uma matriz Dim x Dim
createBoard([H],Dim):-
  length(H, Dim),
  domain(H,0,1).
createBoard([H|Answer], Dim):-
  length(H, Dim),
  domain(H,0,1),
  createBoard(Answer, Dim).
```

numbered_positions_neighbors_constraint.pl Código para restringir as casas vizinhas às células númeradas

```
%verifica se os arredores de uma posicao numerada tem o numero certo de casas pintadas
checkNumberedPositions([], _, _).
checkNumberedPositions([H|R], QueueBoard, Dim):-
  checkAdjacentValues(H, QueueBoard, Dim),
  checkNumberedPositions(R, QueueBoard, Dim).


checkAdjacentValues(X-Y-V, QueueBoard, Dim):-
  getBoardValues(X, Y, QueueBoard, Dim, Elem, Pos, ValueUpLeft, ValueUp,
  ValueUpRight, ValueLeft, ValueRight, ValueDownLeft, ValueDown,
  ValueDownRight),
  Elem #= 0,
  V #= ValueUpLeft + ValueUp + ValueUpRight + ValueLeft + ValueRight +
  ValueDownLeft + ValueDown + ValueDownRight.
```

print.pl Código que desenha o resultado (tabuleiro)

```
%['trace_road.pl'],calculate_road([2-2-5,5-2-5,3-3-3,2-5-3,6-6-2]-[5-4],6,RES),showBoard(RES
/*Predicado Para Mostrar o Tabuleiro*/
showBoard(Solution,Board-Zeros,Dim):- !,createPrintBoard(Solution,Board-Zeros,Dim,X),nl,shou
showRow(X, Y,Dim):- X = [],!.
showRow(X, Y,Dim):- !, X = [H|R],
Yn is (Y + 1),
writeSlash(Dim),
nl,
showRowValues(H), nl,
showRow(R, Yn,Dim).

showRowValues(X):- X = [] ,!, write(' |').
showRowValues(X):- !,X = [H|R],
write(' | '),
showPiece(H),
showRowValues(R).

showPiece(X):- X = 11,!,
put_code(9619).
showPiece(X):- X = 10,!,
write(' ').
showPiece(X):- X = 9,!,
put_code(9209).
showPiece(X):- write(X).

writeSlash(0):-!.
writeSlash(Dim):-
write('----'),
ND is Dim - 1,
writeSlash(ND).

createPrintBoard(Solution,Clues-Zeros,Dim,Square):-
length(Square, Dim),
  createPBoard(Square, Dim),
putZeros(Square,Zeros),
putClues(Square,Clues),
putPath(Solution,Square).

putPathLin([],[]).
putPathLin([S|LinSol],[Q|LinSqu]):-
ite(nonvar(Q),true,(Elem is S + 10,Q is Elem)),
putPathLin(LinSol,LinSqu).

putPath([],[]).
```

```
putPath([LinSol|Solution],[LinSqu|Square]):-
putPathLin(LinSol,LinSqu),
putPath(Solution,Square).

putClues(_,[]).
putClues(Square,[X-Y-V|Clues]):-
nth1(Y,Square,Lin),
nth1(X,Lin,V),
putClues(Square,Clues).

putZeros(_,[]).
putZeros(Square,[X-Y|Zeros]):-
nth1(Y,Square,Lin),
nth1(X,Lin,9),
putZeros(Square,Zeros).
createPBoard([H],Dim):-
length(H, Dim).
createPBoard([H|Answer], Dim):-
length(H, Dim),
createPBoard(Answer, Dim).
```

simple_separated_roads_constraint.pl Código que tenta restringir uma resposta a uma única rua.

```
%verifica a transicao de uma coluna/linha limpa (so casas brancas)
%para uma coluna/linha com pelo menos uma casa preta
checkClearLine([], _, 0).
checkClearLine([Row|OtherRows], State, Result):-
  count(1, Row, #=, Count),
  (Count #>= 1 #/\ State #= 0) #<=> B,
  Result #= ResultAux + B,
  checkClearLine(OtherRows, Count, ResultAux).

checkSimpleSeparatedRoads(Matrix) :-
  checkClearLine(Matrix, 0, Result1),
  transpose(Matrix, TransposeMatrix),
  checkClearLine(TransposeMatrix, 0, Result2),
  Result1 #= 1, Result2 #= 1.
```

test_result.pl Código que testa, depois de criada uma resposta, se é solução.

```
:- dynamic pos/1, index/1.

%falha se houver mais do que um caminho fechado
uniqueRoad(BoardQueue, Dim) :-
  retractall(pos(_)),
  retractall(index(_)),
  assert(pos([])),
  assert(index([1])),
  checkResultValues(BoardQueue, Dim, 1),
  hasAdjacent,
  pos(Pos),
  index(I),
  length(Pos, TotalSize),
  length(I, FinalSize),
  write('TotalSize = '), write(TotalSize), nl,
  write('FinalSize = '), write(FinalSize), nl,
  write('-------------------------------------'), nl, nl,
  retract(pos(_)),
  retract(index(_)),
  TotalSize = FinalSize.

%percorre o tabuleiro resultante
checkResultValues([], _, _).
checkResultValues([H|T], Dim, It) :-
  AuxX is It mod Dim,
  AuxY is It//Dim,
  ite(AuxX = 0, (X is Dim, Y is AuxY), (X is AuxX, Y is AuxY + 1)),
  NewIt is It + 1,
  savePosition(X, Y, H),
  checkResultValues(T, Dim, NewIt).

%guarda a posicao das casas a preto (do caminho)
savePosition(X, Y, 0).
savePosition(X, Y, 1) :-
  retract(pos(Pos)),
  append(Pos, [X-Y], NewPos),
  assert(pos(NewPos)).

hasAdjacent :-
  index(IndexList), pos(Pos),
  nth1(1, IndexList, I),
  nth1(I, Pos, X-Y),
  LeftX is X - 1, RightX is X + 1,
  UpY is Y - 1, DownY is Y + 1,
```

```
  ite((member(LeftX-Y, Pos), nth1(AdjLeft, Pos, LeftX-Y),
  \+member(AdjLeft, IndexList)),
    (retract(index(_)), append([AdjLeft], IndexList, NewIndexList),
    assert(index(NewIndexList)), hasAdjacent),
    ite((member(X-UpY, Pos), nth1(AdjUp, Pos, X-UpY), \+member(AdjUp, IndexList)),
      (retract(index(_)), append([AdjUp], IndexList, NewIndexList),
      assert(index(NewIndexList)), hasAdjacent),
      ite((member(RightX-Y, Pos), nth1(AdjRight, Pos, RightX-Y),
      \+member(AdjRight, IndexList)),
        (retract(index(_)), append([AdjRight], IndexList, NewIndexList),
        assert(index(NewIndexList)), hasAdjacent),
        ite((member(X-DownY, Pos), nth1(AdjDown, Pos, X-DownY),
        \+member(AdjDown, IndexList)),
          (retract(index(_)), append([AdjDown], IndexList,
          NewIndexList), assert(index(NewIndexList)), hasAdjacent),
          true)
        )
      )
    ).
```

statistic.pl Código do ficheiro sobre estatísticas.

```
reset_timer :- statistics(walltime,_).
print_time :-
statistics(walltime,[_,T]),
TS is ((T//10)*10)/1000,
nl, write('Time: '), write(TS), write('s'), nl, nl.
```