Four Winds:
Resolução de Problemas de Decisão com recurso a Programação Lógica com Restrições

Ana Rita Torres e Rui Pedro Soares

Turma:3MIEIC06**, Grupo:** Four_Winds_2
Faculdade de Engenharia da Universidade do Porto,
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal
{up201406093, up201404965}@fe.up.pt
http://www.fe.up.pt

Resumo Este artigo completa a realização do segundo projeto da Unidade Curricular de Programação em Lógica do Mestrado Integrado de Engenharia Informática e Computação. O projeto tem como objetivo principal o desenvolvimento da solução para o jogo *Four Winds* escrita em Prolog, apresentado este um problema de decisão. A abordagem usada permite lidar com tabuleiros de diversas dimensões, assim como com um número de peças variável. É também implementada uma solução do problema sob a forma de texto de modo a permitir a visualização desta.

Palavras-Chave: prolog, four winds, solução, decisão

1 Introdução

O objetivo deste projeto era implementar a resolução de um problema de decisão ou otimização em Prolog com restrições.
O grupo optou por um problema de decisão, em específico, o puzzle Four Winds. O puzzle apresenta uma estrutura quadrangular e o seu preenchimento é realizado através de traços e setas.
Este artigo visa descrever de forma detalhada o problema; a abordagem utilizada pelo grupo para a resolução deste; a explicação da representação utilizada para as soluções; as estatísticas de resolução de puzzles de diferentes dimensões; conclusões relativas ao projeto, no geral.

2 Descrição do Problema

O puzzle Four Winds apresenta uma estrutura quadrangular, isto é, tem o mesmo número de linhas e colunas. Um tabuleiro inicial tem o seguinte aspeto.

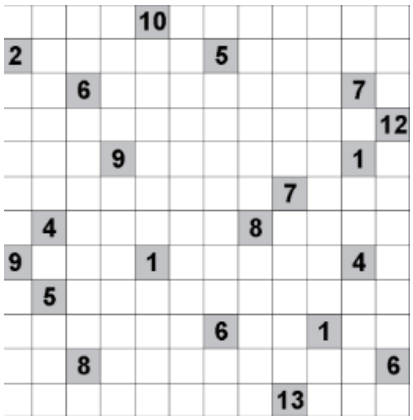

Figura 1 - Puzzle Inicial(12x12)

Desenha-se uma ou mais linhas a partir de cada célula numerada de forma a que cada número indique o comprimento total de linhas que são desenhadas a partir dessa célula, excluindo a própria célula. As linhas podem ser, apenas, verticais ou horizontais. Estas podem ligar centros de células adjacentes, mas sempre sem se sobrepor ou cruzar entre elas e, relativamente às células também. Não há qualquer tipo de limitação relativo à repetição dos valores das células numeradas.
Para facilitar a familiarização com as regras do puzzle, segue-se o tabuleiro anterior resolvido.

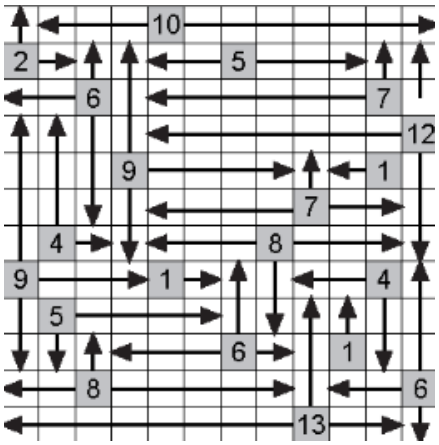

Figura 2 - Puzzle Resolvido(12x12)

3 Abordagem

Tal como muitos outros dos outros projetos à escolha, este trata-se de um problema de obter áreas, no tabuleiro, que se adequem à solução. Como tal, abordámos o problema com a seguinte resolução em mente:

1. Cada número no tabuleiro inicial corresponde à área de uma região, excluindo a célula onde se encontra. O primeiro passo consiste em identificar essa região com um índice.
2. Obter todos os tuplos (Índice da Região, Área da Região, Coordenada X da capital, Coordenada Y da capital).
3. Inicializar um tabuleiro com variáveis, e colocar os índices das capitais no sítio correto.
4. Aplicar restrições às restantes células que não sejam capitais.
5. Obter a solução possível, através de *labeling*.

Como tal, um tabuleiro como o seguinte

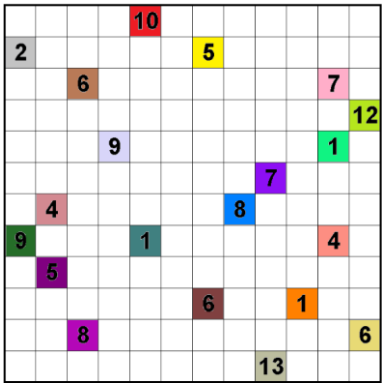

Figura 3 - Puzzle Inicial Exemplo

, deverá resultar na solução possível.

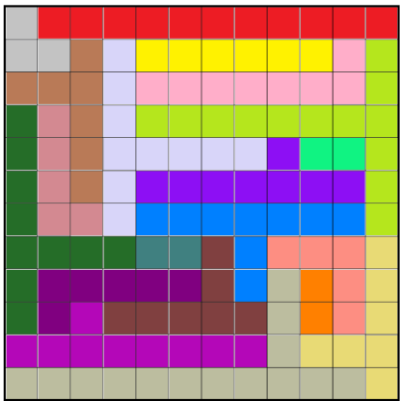

Figura 4 - Regiões

A solução obtida para cada célula consiste no índice da região que a ocupa. No exemplo, a célula (1,0) teria índice 0, correspondente à região da capital em (4,0), com valor de área igual a 10.

O predicado que trata de resolver o puzzle é ***solver(Puzzle, Solution)***, sendo que Puzzle é o tabuleiro (em formato Prolog) inicial, e Solution a matriz-solução. Solution pode dar mais que um resultado, consoante o puzzle escolhido.
É de notar que a interface criada (predicado fourWinds) mostra-nos apenas uma das potenciais múltiplas soluções.

Para além disto, achamos relevante referir que enquanto a nossa solução é uma matriz, o predicado *labeling* necessita de uma lista simples.
Para resolver isto, recorremos ao "flattening" da matriz-solução, ou seja, convertê-la numa lista simples (preservando todas as restrições, domínios, etc. aplicados) e só depois aplicando *labeling*.
Depois disto, voltamos a convertê-la numa matriz, para ser mais intuitiva de imprimir no ecrã.

3.1 Variáveis de Decisão

A solução tem NxN variáveis de decisão totais (uma para cada célula).
Cada variável deve ter como seu domínio a região possível que pode ocupar a respetiva casa.
Inicialmente, é-lhes dado o domínio de 0 (índice mínimo) a Z, sendo Z o número de regiões a serem criadas (obtido facilmente, porque cada capital representa sempre uma região, logo Z é o número de capitais).

Depois, é-lhes restrito o domínio de acordo com as regiões que podem ocupar a célula. Cada casa pode no máximo ter 4 regiões (cima, baixo, esquerda, direita) no seu domínio.

Na geração da solução (em labeling), testa-se todos os valores possíveis de variáveis de decisão.

3.2 Restrições

Organizámos as principais restrições do puzzle em três predicados:

- ***constrainCells***: cada célula pode no máximo ter no domínio da sua variável de decisão correspondente 4 regiões possíveis.
  Esta função vai, de célula a célula, restringir o domínio de cada variável para as possíveis casas.
  Isto é feito verificando no tabuleiro inicial, para as coordenadas da célula atual, as regiões imediatamente (excluindo espaços vazios) acima, abaixo, à esquerda e à direita da célula. Estas serão as únicas regiões passíveis de pertencer ao domínio da variável de decisão da célula, e como tal, restringe-se para tal.

- ***constrainAreas***: este é essencialmente o grande problema do puzzle: fazer com que as regiões tenham áreas correspondentes às indicadas pelas capitais.
  Isto é feito obtendo a área da região, e restringindo-a para o indicado na capital. Isto exclui todas os resultados em que uma área tivesse mais células ocupadas do que o indicado.

- ***constrainCohesions***: embora as outras restrições sejam as mais fundamentais, não chegam, pois podiam levar a casos onde a área de cada região é efetivamente respeitada, mas as regiões ficariam divididas pelas outras regiões. Este predicado obriga a que o total de células a partir de cada capital seja o da área indicada, e portanto, que a região esteja coesa.

3.3 Estratégia de Pesquisa

Usámos a estratégia de etiquetagem *bisect*. No entanto, as outras opções de etiquetagem não mudam significativamente a performance do programa, devido à natureza da solução pela qual optámos, e pela quantidade de restrições aplicadas.

4 Visualização da Solução

Para iniciar o programa é necessário inserir: "fourWinds. "na consola e o utilizador irá deparar-se com o seguinte menu.

```
------------------------- FOUR WINDS : A BOARD PUZZLE -------------------------
-------------------------------------------------------------------------------

Press the following to start:
1 - Run puzzle solver
2 - Read about Four Winds
3 - Quit

|:
```

Figura 5 - Menu Principal

Como se pode verificar este dá-nos três opções:

1. Ir para o solucionador de puzzles;
2. Ler sobre o puzzle;
3. Sair do jogo.

Ao selecionar o "1." o utilizador é redirecionado para o ecrã abaixo apresentado.

```
------------------------- FOUR WINDS : A BOARD PUZZLE -------------------------
-------------------------------------------------------------------------------

Choose a board side size (?x?, from 2x2 to 12x12)
|:
```

Figura 6 - Escolha da dimensão do Tabuleiro

Aqui o utilizador pode escolher o tamanho do tabuleiro, entre os valores apresentados, que quer ver solucionado. Para efeitos de exemplificação, suponha-se que o utilizador optou por um tabuleiro de tamanho 5. A impressão deste é realizada pelo predicado ***printUnsolvedPuzzle*** este recebe o puzzle sob a forma de matriz, uma matriz que é composta por números, no caso da célula se encontrar vazia -1, se se encontrar numerada estará lá representado esse valor.

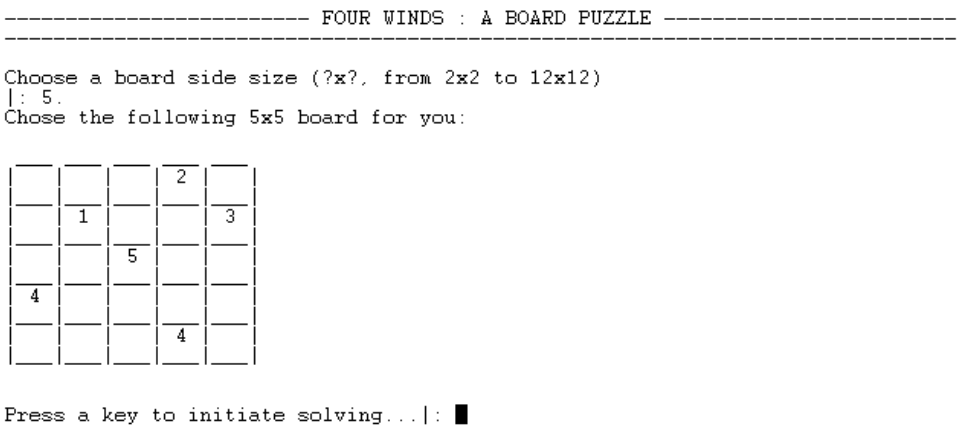

Figura 7 - Tabuleiro Sem Solução

Esta imagem representa um tabuleiro do tamanho inserido pelo utilizador no seu estado inicial e clicando em qualquer tecla é-lhe apresentada a solução, assim como as estatísticas relativas ao puzzle. A impressão do tabuleiro é realizada pelo predicado ***printSolution*** que recebe a solução do puzzle e a lista com as coordenadas das células numeradas, assim como mapeamento das suas regiões. Este predicado também se encarrega da impressão das setas, dos traços e dos números apresentados.

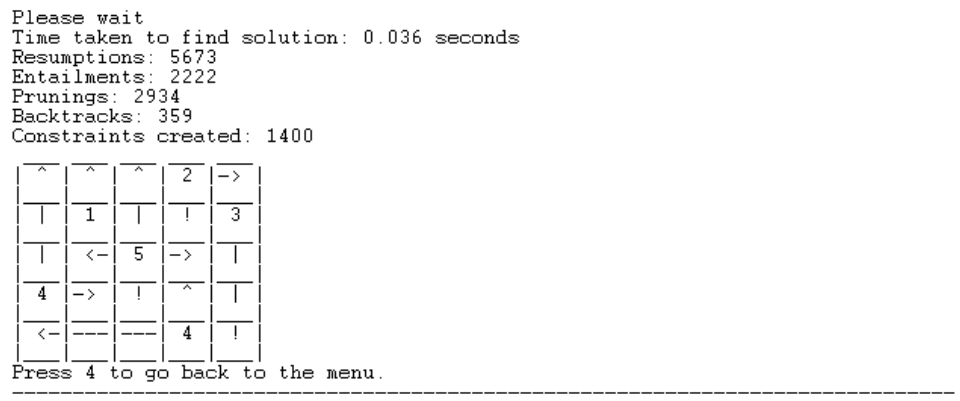

Figura 8 - Solução

Como se pode ver no ponto 2, o jogo usa setas para indicar a direção em que as linhas são desenhadas as representações horizontais pareceram lógicas ao grupo por outro lado, as representações verticais implicaram alguma criatividade e

como tal o acento circunflexo (^) é usado para representar uma seta para cima e o ponto de exclamação (!) a seta para baixo.
Os predicados desenvolvidos para a visualização do tabuleiro encontram-se no ficheiro printer.pl.

5 Resultados

Tamanho do tabuleiro	Tempo	Resumptions	Entailments	Prunings	Backtracks
2x2	< 0,001	567	271	229	278
3x3	0,001	815	294	403	57
4x4	0,002	1756	627	830	157
5x5	0,004	3700	1315	1789	320
6x6	0,007	4339	3370	4586	588
...	...	...	...	...	...
12x12	15,5	63839663	21038451	30622427	1195971

Figura 9 - Estatísticas de Performance

A performance do código desenvolvido revelou-se pouco eficiente em puzzles que apresentam dimensões maiores. Os resultados apresentados na tabela não são 100% fiáveis, uma vez que usamos uma amostra pequena (cerca de 4 a 5 puzzles por dimensão), consequência de não ter sido concebido um gerador de puzzles.

6 Conclusões e Trabalho Futuro

Este projeto foi essencial para a consolidação de conceitos relativos à Programação em Lógica com restrições, na medida em que foram colocados à prova os conhecimentos do grupo.
Sem dúvida a maior surpresa durante o desenvolvimento do projeto foi o desafio que a representação visual da solução nos colocou, que acreditamos ter resolvido com sucesso.
A solução implementada revelou-se bastante eficaz para tabuleiros de dimensões inferiores a 12, no caso de ser superior a eficácia está dependente do número de soluções para o puzzle proposto, quanto mais soluções existem, mais lento é o processo.

Bibliografia

1. Regras do Four Winds, http://www.worldpuzzle.org/championships/types-of-puzzles/wpc/
2. SWI-Prolog, http://www.swi-prolog.org
3. SICStus Prolog, http://sicstus.sics.se/

Anexo

Código Fonte

proj2.pl

```
:- [    'solver.pl',
        'matrixRep.pl',
        'printer.pl',
        'interface.pl'
].
```

interface.pl

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

clr:- write('\33\[2J').

fourWinds :-
repeat,
clr,
nl,nl,
write('------------------------- FOUR WINDS : A BOARD PUZZLE -------------------------'),
nl,
write('--------------------------------------------------------------------------------'), nl, nl,
write('Press the following to start: '), nl,
write('1 - Run puzzle solver'), nl,
write('2 - Read about Four Winds'), nl,
write('3 - Quit'), nl, nl,
```

```
read(Input),
menu(Input).

menu(1) :-
repeat,
clr,
nl,nl,
write('------------------------- FOUR WINDS : A BOARD PUZZLE -------------------------'),
nl,
write('-------------------------------------------------------------------------------'), nl, nl,
write('Choose a board side size (?x?, from 2x2 to 12x12)'), nl,
read(Input),
getAPuzzleOfSize(Input, Puzzle),
write('Chose the following '), write(Input), write('x'), write(Input), write(' board
for you: '), nl, nl,
printUnsolvedPuzzle(Puzzle), nl, nl,
write('Press a key to initiate solving...'),
get_char(_), get_char(_),
nl, write('Please wait'), nl,
solvePuzzle(Puzzle),
!,
write('Press 4 to go back to the menu.'), nl,
write('-------------------------------------------------------------------------------'), nl, nl,
read(Input),
menu(Input).

menu(2) :-
repeat,
clr,
nl, nl,
write('------------------------- FOUR WINDS : A BOARD PUZZLE -------------------------'),
nl,
write('-------------------------------------------------------------------------------'), nl, nl,
write('Draw one or more lines from each numbered cell so that each number'), nl,
write('indicates the total length of lines that are drawn from that cell,'), nl,
write('excluding the cell itself.'), nl, nl,
write('Lines are either horizontal or vertical and connect the centers of '), nl,
write('adjacent cells without crossing or overlapping each other'), nl, write('and
the given numbers.'), nl, nl,
write('Press 4 to go back to the menu.'), nl,
write('-------------------------------------------------------------------------------'), nl, nl,
read(Input),
menu(Input).
menu(3) :- clr.
```

```
menu(4) :- fourWinds.

getPuzzlesOfSize(Size, Puzzles) :-
findall(Board, puzzle(Size, Board), Puzzles).

getAPuzzleOfSize(Size, Puzzle) :-
getPuzzlesOfSize(Size, Puzzles),
length(Puzzles, N),
random(0, N, X),
nth0(X, Puzzles, Puzzle).

solvePuzzle(Board) :-
solver(Board, Result),
getIndexedNumbers(Board, Indexed),
printSolution(Result-Indexed), !.
```

matrixRep.pl

```
puzzle(12, [[-1,-1,-1,-1,10,-1,-1,-1,-1,-1,-1,-1],
                    [2,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,-1],
                    [-1,-1,6,-1,-1,-1,-1,-1,-1,-1,7,-1],
                    [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,12],
                    [-1,-1,-1,9,-1,-1,-1,-1,-1,-1,1,-1],
                    [-1,-1,-1,-1,-1,-1,-1,-1,7,-1,-1,-1],
                    [-1,4,-1,-1,-1,-1,-1,8,-1,-1,-1,-1],
                    [9,-1,-1,-1,1,-1,-1,-1,-1,-1,4,-1],
                    [-1,5,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1],
                    [-1,-1,-1,-1,-1,-1,6,-1,-1,1,-1,-1],
                    [-1,-1,8,-1,-1,-1,-1,-1,-1,-1,-1,6],
                    [-1,-1,-1,-1,-1,-1,-1,-1,-1,13,-1,-1,-1]]).

puzzle(12, [ [-1,-1,-1,14,-1,-1,-1,-1,-1,-1,-1,-1],
                                [-1,-1,-1,-1,-1,-1,-1,-1,-1,10,-1,-1,-1],
```

```
                    [-1,-1,2 ,-1,-1,2 ,-1,-1,-1,5 ,-1,-1],
                    [-1,-1,-1,-1,-1,3 ,-1,-1,-1,-1,-1,-1],
                    [-1,6 ,-1,-1,-1,-1,-1,4 ,-1,-1,-1,-1],
                    [-1,-1,-1,-1,11,-1,-1,-1,-1,-1,-1,-1],
                    [10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,3 ,-1],
                    [-1,-1,-1,-1,-1,-1,2 ,-1,-1,-1,-1,13],
                    [-1,-1,3 ,-1,-1,9 ,-1,-1,-1,-1,-1,-1],
                    [-1,-1,-1,-1,-1,-1,5 ,-1,-1,-1,-1,-1],
                    [-1,-1,-1,6 ,-1,-1,-1,-1,-1,-1,5 ,-1],
                    [-1,7 ,-1,-1,-1,-1,-1,-1,4 ,-1,-1,-1]]).


puzzle(12, [ [-1,-1,-1,3 ,-1,-1,-1,-1,-1,-1,4 ,-1],
                    [-1,-1,-1,-1,-1,-1,-1,2 ,-1,-1,-1,-1],
                    [-1,3 ,-1,-1,-1,-1,4 ,-1,-1,-1,-1,-1],
                    [-1,-1,-1,-1,-1,6 ,-1,-1,-1,-1,-1,10],
                    [-1,-1,10,-1,-1,-1,-1,-1,-1,13,-1,-1],
                    [-1,-1,-1,-1,11,-1,-1,-1,-1,-1,-1,-1],
                    [11,-1,-1,-1,-1,-1,-1,-1,6 ,-1,-1,-1],
                    [-1,-1,-1,5 ,-1,-1,-1,-1,-1,-1,-1,-1],
                    [-1,-1,-1,-1,-1,-1,3 ,-1,-1,-1,-1,-1],
                    [-1,10,-1,-1,-1,-1,-1,-1,-1,-1,4 ,-1],
                    [-1,-1,-1,-1,3 ,-1,-1,8 ,-1,-1,-1,-1],
                    [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,9 ,-1,-1]]).

puzzle(12, [ [-1,-1,5 ,-1,-1,-1,-1,-1,-1,6 ,-1,-1],
                    [-1,-1,-1,-1,9 ,-1,-1,-1,-1,-1,-1,-1],
                    [-1,5 ,-1,-1,-1,-1,7 ,-1,-1,-1,3 ,-1],
                    [4 ,-1,-1,-1,-1,0 ,-1,-1,4 ,-1,-1,-1],
                    [-1,-1,-1,3 ,-1,-1,-1,-1,-1,-1,-1,8 ],
                    [-1,-1,-1,-1,-1,-1,-1,3 ,-1,-1,0 ,-1],
                    [-1,0 ,-1,-1,-1,9 ,-1,-1,-1,-1,-1,-1],
                    [-1,-1,6 ,-1,-1,-1,-1,-1,-1,2 ,-1,-1],
                    [6 ,-1,-1,-1,-1,-1,0 ,-1,6 ,-1,-1,-1],
                    [-1,-1,-1,-1,5 ,-1,-1,-1,-1,-1,8 ,-1],
                    [-1,-1,-1,4 ,-1,-1,-1,3 ,-1,-1,-1,-1],
                    [-1,4 ,-1,-1,-1,-1,-1,-1,-1,-1,-1,8 ]]).

puzzle(11, [        [-1,-1,-1,-1,-1,-1,-1, 8,-1,-1,-1],
                    [ 6,-1,-1,-1,-1,-1,-1,-1, 2,-1,-1],
                    [ 2,-1,-1,-1,-1, 5,-1,-1,-1,-1,-1],
                    [-1,-1, 1,-1,-1, 2,-1,-1,-1,12,-1],
                    [-1,-1, 2,-1,-1, 1,-1,-1,-1,-1,-1],
```

```
                    [ 1,-1,-1,-1, 7,-1,-1,-1,-1,-1,10],
                    [-1, 2,-1,-1,-1,-1,-1, 4,-1,-1,-1],
                    [-1,-1,-1, 6,-1,-1,-1, 3,-1,-1,-1],
                    [ 3,-1, 0,-1, 1,-1,-1,-1,-1,-1,-1],
                    [-1,-1,-1,-1,-1,-1, 6,-1,-1,-1,-1],
                    [-1,-1,-1,-1,-1,-1,-1,-1,15,-1,-1]]).

puzzle(10, [    [3 ,-1,1 ,-1,4 ,-1,3 ,-1,4 ,-1],
                    [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1],
                    [-1,3 ,-1,2 ,-1,4 ,-1,2 ,-1,4 ],
                    [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1],
                    [2 ,-1,2 ,-1,2 ,-1,1 ,-1,2 ,-1],
                    [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1],
                    [-1,5 ,-1,5 ,-1,5 ,-1,4 ,-1,4 ],
                    [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1],
                    [3 ,-1,2 ,-1,3 ,-1,2 ,-1,3 ,-1],
                    [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]]).

puzzle(9, [             [-1,-1,-1, 5,-1,-1,-1,-1,-1],
                    [ 2,-1, 5,-1,-1,-1,-1, 3,-1],
                    [-1,-1,-1, 2,-1,-1,-1,-1,-1],
                    [-1, 3,-1,-1,-1,-1, 3,-1,-1],
                    [-1,-1,-1,-1, 3,-1,-1,-1,-1],
                    [-1,-1, 4,-1,-1,-1,-1,-1,-1],
                    [ 6,-1,-1,-1,-1, 5,-1,-1, 6],
                    [-1,-1,-1, 4,-1,-1,-1, 7,-1],
                    [-1, 2,-1,-1, 5,-1,-1,-1,-1]]).

puzzle(8, [[-1,6,-1,-1,-1,-1,-1,-1],
                [14,-1,-1,-1,-1,-1,-1,-1],
                [-1,2,-1,-1,-1,-1,-1,3],
                [-1,2,-1,-1,-1,-1,-1,3],
                [-1,-1,-1,6,-1,-1,-1,-1],
                [-1,4,-1,-1,-1,2,-1,-1],
                [-1,-1,4,-1,-1,-1,-1,2],
                [-1,-1,-1,4,-1,-1,-1,-1]]).

puzzle(7, [     [-1,-1,-1, 3, 1,-1,-1],
                    [ 1,-1, 0,-1, 1, 1, 2],
                    [-1, 5,-1,-1,-1,-1,-1],
                    [-1,-1,-1,-1, 2,-1,-1],
                    [ 1,-1, 5,-1,-1,-1,-1],
```

```
                    [-1, 4,-1,-1,-1,-1, 3],
                    [-1,-1,-1, 6,-1,-1,-1] ]).

puzzle(6, [     [-1,-1,6 ,-1,-1,-1],
                    [-1,4 ,-1,-1,-1,2 ],
                    [-1,-1,-1,1 ,-1,-1],
                    [-1,-1,2 ,-1,-1,-1],
                    [3 ,-1,-1,-1,6 ,-1],
                    [-1,-1,-1,4 ,-1,-1] ]).

puzzle(6, [     [ 9,-1,-1,-1,-1,-1],
                    [-1,-1, 2,-1,-1,-1],
                    [-1,-1,-1,-1, 4,-1],
                    [-1,-1,-1,-1,-1, 9],
                    [-1,-1, 3,-1,-1,-1],
                    [-1, 1,-1,-1, 1,-1] ]).

puzzle(6, [     [0 ,-1,-1,-1, 1,-1],
                    [-1, 2,-1,-1,-1, 3],
                    [-1,-1,10,-1,-1,-1],
                    [-1, 2,-1, 2,-1,-1],
                    [ 2,-1,-1,-1,-1, 3],
                    [-1,-1,-1,-1, 1,-1] ]).

puzzle(5,  [    [4, -1, -1, -1,  -1],
                    [-1, 5, -1, -1, -1],
                    [2, -1, 2 , -1, -1],
                    [-1, 3, -1, -1, -1],
                    [-1, -1, -1, -1, 3] ]).

puzzle(5,  [    [-1,-1,-1,2 ,-1],
                    [-1,1 ,-1,-1,3 ],
                    [-1,-1,5 ,-1,-1],
                    [4 ,-1,-1,-1,-1],
                    [-1,-1,-1,4 ,-1] ]).

puzzle(5,  [    [-1,-1,-1,6 ,-1],
                    [5 ,-1,-1,-1,-1],
                    [-1,-1,4 ,-1,-1],
                    [-1,-1,-1,-1,4 ],
                    [-1,1 ,-1,-1,-1] ]).

puzzle(5,  [    [-1, 1,-1, 1,-1],
```

```
                    [-1, 4, 4, 4,-1],
                    [-1,-1,-1,-1,-1],
                    [ 2,-1,-1,-1, 2],
                    [-1,-1,-1,-1,-1] ]).

puzzle(5, [         [-1, 3,-1,-1, 2],
                    [ 3,-1, 3,-1,-1],
                    [-1,-1,-1, 3,-1],
                    [-1, 1,-1,-1, 1],
                    [-1,-1, 1,-1,-1] ]).

puzzle(4,           [[-1,4 ,-1,-1],
                    [-1,-1,0 ,-1],
                    [-1,-1,-1,4 ],
                    [4 ,-1,-1,-1]]).

puzzle(4,           [[-1,4 ,-1,-1],
                    [2 ,-1, 1,-1],
                    [-1, 3,-1,-1],
                    [-1,-1, 1,-1]]).

puzzle(4,           [[ 0,-1,-1, 0],
                    [-1, 2, 2,-1],
                    [-1, 2, 2,-1],
                    [ 0,-1,-1, 0]]).

puzzle(4,           [[-1,-1,-1,-1],
                    [-1,-1, 1,-1],
                    [-1, 3,-1, 2],
                    [6,-1,-1,-1]]).

puzzle(3, [[ 4, -1, -1],
           [ -1, 2, -1],
           [ -1, -1, 0]]).

puzzle(3, [[ 0, -1, 0],
           [ -1, 4, -1],
           [ 0, -1, 0]]).

puzzle(3, [[ 2, -1, -1],
           [ -1, -1, 3],
           [ -1, 1, -1]]).
```

```
puzzle(3, [[ -1, 2, -1],
                    [ 1, -1, -1],
                    [ -1, -1, 3]]).

puzzle(2,  [[2, -1],
                    [-1, 0]]).

puzzle(2,  [[1, -1],
                    [-1, 1]]).

puzzle(2, [[0,0],
                    [0,0]]).
```

solver.pl

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


% This is the predicate that runs the algorithm (restrictions) to solve the puzzle.

% Our solution involves separating the puzzle board into smaller regions.
% Each region corresponds to a number in the main puzzle, ie, the number 1
% would have an arrow up, down, left or right of it. We don't need the direction
% for the solution. Simply see that the length specified by the number + 1
% (this 1 is the square occupying the number) equals the total area of the region!
% Then, we just ID the region and see all the possible regions with restrictions...

% An example of an original board and the generated solution:
% -1  4 -1 -1                    0 0 0 0
% -1 -1  0 -1                    3 0 1 2          This means the 4 on coordinates (x=1, y=0) became the capital of region 0, which spreads
% -1 -1 -1  4          -->       3 2 2 2          as shown. The region's area is 5, which is the original 4 + 1.
%  4 -1 -1 -1                    3 3 3 2
```

```
solver(Board, Solution) :-

% board is always square; calculate its side.
length(Board, N),

% see where the numbers are. Assign them to a region (give them an index).
getIndexedNumbers(Board, Indexed),
length(Indexed, MaxRegions1),
MaxRegions #= MaxRegions1 - 1,

% create an empty matrix, to be restricted
createEmptySolution(N, MaxRegions, EmptyResult),

% initiate statistics
statistics(walltime, _),

% start off the regions where the numbers are on the original board (these are
called capitals). Place the capital index on the matrix, not the length.
initRegions(EmptyResult, Indexed, Result),

% See that cells above, below, left and right of these starting regions may only be
occupied by these regions.
% Ie, a cell that's diagonal to a capital can never be on that capital's region.
constrainCells(Board, Result),

% The true problem of "Four Winds". Constrain the board so that it creates regions
with areas said by the numbers.
constrainAreas(Indexed, Result, 0, MaxRegions1),

% The last two constraints would have made disjoint regions of the same index
possible... This one takes those out of the solution, though.
constrainCohesions(Indexed, Result, 0, MaxRegions1),

% We were opperating on a matrix because it was a lot easier to visualize and
manipulate. To generate the solution, we need to flatten out the matrix on a single
list.
flatten_list(Result, Flat_Solution),

% Generate solution
labeling([bisect], Flat_Solution),

% Obtain and print statistics
statistics(walltime, [_, Elapsed | _]),
```

```
format('Time taken to find solution: ~3d seconds', Elapsed), nl,
fd_statistics,

% Convert back, so it's easier to print.
list_to_matrix(Flat_Solution, N, Solution).


%%%%%%%%%%%%%%%%%%%%%%%%%%% AUXILARY PREDICATES
%%%%%%%%%%%%%%%%%%%%%%%%%%%

% This predicate gets a matrix's element at a given pair of coordinates.
getNumberAtCoord(-1,-1,-1,_).
getNumberAtCoord(X, Y, Number, Board) :-
nth0(Y, Board, Row),
nth0(X, Row, Number).

% This predicate sets a matrix's element at a given pair of coordinates.

setNumberAtCoord(X, Y, Number, BoardIn, BoardOut) :-
        setLineAux(0,X,Y,Number,BoardIn,BoardOut), !.

setLineAux(_,_,_,_,[],[]).
setLineAux(Y, X, Y, Number, [Line | Tail], [Line2 | Tail2]) :-  setColAux(0,
X, Number, Line, Line2),

                                                Ynow is Y + 1,

                                                setLineAux(Ynow, X, Y,
Number, Tail, Tail2).
setLineAux(Ynow, X, Y, Number, [Line | Tail], [Line | Tail2]) :-  Ynow \= Y,

                                                        Ynow2 is Ynow + 1,

                                                        setLineAux(Ynow2, X,
Y, Number, Tail, Tail2).

setColAux(_,_,_,[],[]).
setColAux(X, X, Number, [ _ | Tail], [Number | Tail2]) :-  Xnow is X + 1,

                                                setColAux(Xnow, X, Number, Tail,
Tail2).

setColAux(Xnow, X, Number, [Element | Tail], [Head | Tail2]) :-  Xnow \= X,
```

```
                                        Xnow2 is Xnow + 1,
                                        Head #= Element,
                                        setColAux(Xnow2, X,
Number, Tail, Tail2).


% This predicate gets all the puzzle's numbered squares (capitals), coupled with
their (X, Y) coordinates.

getNumbers(Board, Numbers) :-
length(Board, N),
getNumbersInRows(Board, 0, N, List),
flatten_list(List, Numbers).

getNumbersInRows(_, N, N, []).
getNumbersInRows(Board, Y, N, [Head | Tail]) :-
nth0(Y, Board, Row),
getNumbersInRow(Row, 0, Y, N, Head),
Y1 is Y + 1,
getNumbersInRows(Board, Y1, N, Tail).

getNumbersInRow(_, N, _, N, []).
getNumbersInRow(Row, X, Y, N, [Head | Tail]) :-
nth0(X, Row, Number),
Number \= -1,
Head = [Number, X, Y],
X1 is X + 1,
getNumbersInRow(Row, X1, Y, N, Tail).
getNumbersInRow(Row, X, Y, N, Tail) :-
nth0(X, Row, Number),
Number == -1,
X1 is X + 1,
getNumbersInRow(Row, X1, Y, N, Tail).

% Auxiliary Predicate to flatten the list

flatten_list([], []).
flatten_list( [HeadList| TailList], Result) :-
flatten_list( TailList, NewTailList),
!,
append(HeadList, NewTailList, Result).
```

```
flatten_list( [ HeadList | Tail ], [ HeadList | OtherTail ]) :-
flatten_list( Tail, OtherTail).

% A predicate to add an index to each member of the squares list
% Ie, this gets all the information we need about each capital: [Region_Index, Region_TotalArea, Region_CapitalX, Region_CapitalY].

indexNumbers(Numbers, Indexed) :-
indexNumbersAux(Numbers, 0,Indexed).
indexNumbersAux([], _, [] ).
indexNumbersAux([ NumbersHead | OtherNumbers ], Index, [IndexedHead | IndexedTail] ) :-
append([Index], NumbersHead, IndexedHead),
Index1 is Index + 1,
indexNumbersAux(OtherNumbers, Index1, IndexedTail).

getIndexedNumbers(Board, Indexed) :-
getNumbers(Board, Numbers),
indexNumbers(Numbers, Indexed).

% creates a NxN list (the empty solution). Also, constrain the domain from minRegion (always 0) to maxRegion (number of capitals).

createEmptySolution(N, MaxRegions, Result) :-
createEmptySolutionAux(N, N, Result),
constrainDomain(MaxRegions, Result).

createEmptySolutionAux(_, 0, []).
createEmptySolutionAux(N, NN, [Head | Tail]) :-
length(Head, N),
NN > 0,
NN1 is NN - 1,
createEmptySolutionAux(N, NN1, Tail).

constrainDomain(_,[]).
constrainDomain(MaxRegions, [Head | Tail]) :-
domain(Head, 0, MaxRegions),
constrainDomain(MaxRegions, Tail).

% This is the basic frame for the regions to be constructed upon.

initRegions(Board, [], Board).
initRegions(Board, [[CurrIndex, _, X, Y] | Tail ], Result) :-
```

```
setNumberAtCoord(X, Y, CurrIndex, Board, NewBoard),
initRegions(NewBoard, Tail, Result).

% get a list that, for a cell, gets the first obstacle in each vertical or horizontal direction
% [[VU, VU_X, VU_Y], [VD, VD_X, VD_Y], [HL, HL_X, HL_Y], [HR, HR_X, HR_Y]]. Ie, the region VU, obstructs our cell at coordinates (VU_X, VU_Y). Same for all the other ones...
% If our cell already belongs to a region, it gets all obstacles to that region.

getObstacles(X, Y, Board, List) :-
getNumberAtCoord(X, Y, _, Board),

% Up
getObstacle(X, Y, 0, -1, Board, ObstacleUp),

% Down
getObstacle(X, Y, 0, 1, Board, ObstacleDown),

% Left
getObstacle(X, Y, -1, 0, Board, ObstacleLeft),

% Right
getObstacle(X, Y, 1, 0, Board, ObstacleRight),

List = [ObstacleUp, ObstacleDown, ObstacleLeft, ObstacleRight].

% Up
getObstacle(_, -1, 0, -1, _, [-1,-1,-1]).

% Down
getObstacle(_, Y, 0, 1, Board, [-1,-1,-1]) :-
length(Board, Y).

% Left
getObstacle(-1, _, -1, 0, _, [-1,-1,-1]).

% Right
getObstacle(X, _, 1, 0, Board, [-1,-1,-1]) :-
length(Board, X).

% This predicate gets the obstacle. IF current == -1, stop! Else, continue.
```

```
getObstacle(X, Y, _, _, Board, Obstacle) :-

getNumberAtCoord(X, Y, NumberNow, Board),

NumberNow #\= -1,
!,
Obstacle = [NumberNow, X, Y].

getObstacle(X, Y, IncX, IncY, Board, Obstacle) :-

getNumberAtCoord(X, Y, _, Board),

Xnow #= X + IncX,
Ynow #= Y + IncY,

getObstacle(Xnow, Ynow, IncX, IncY, Board, Obstacle).


% This is predicate that thins down the regions for any cells.
% A cell can belong to any region above, below, left, and right of it (if they exist).
% This thins down cells in the sense that it restrits the cell's domain to only accept
those 4 (max) possible regions.
% If nothing is obstructing the cell on any given side, that side does not interfere
with the domain.

constrainCells(Board, Regions) :-
length(Board, N),
constrainRows(Board, Regions, 0, N).
constrainRows(_, _, N,N).
constrainRows(Board, Regions, Y, N) :-
constrainCols(Board, Regions, 0, Y, N),
Y1 #= Y + 1,
constrainRows(Board, Regions, Y1, N).
constrainCols(_,_, N, _, N).
constrainCols(Board, Regions, X, Y, N) :-
constrainCell(X, Y, Board, Regions),
X1 #= X + 1,
constrainCols(Board, Regions, X1, Y, N).

constrainCell(X, Y, Board, Regions) :-
getObstacles(X, Y, Board, [[_, OU_X, OU_Y], [_, OD_X, OD_Y], [_ ,OL_X, OL_Y], [_, OR_X,
OR_Y]]),
```

```
getNumberAtCoord(OU_X, OU_Y, RegionUP, Regions),
getNumberAtCoord(OD_X, OD_Y, RegionDOWN, Regions),
getNumberAtCoord(OL_X, OL_Y, RegionLEFT, Regions),
getNumberAtCoord(OR_X, OR_Y, RegionRIGHT, Regions),

getNumberAtCoord(X,Y,Var,Regions),

Var in {RegionUP,RegionDOWN,RegionLEFT,RegionRIGHT},
Var #>= 0.

% This is the predicate that checks that all the regions on the board are acceptable
(in area).
% The area is the number given on the original board plus one (counting the
capital as part of the region!).
% It checks the cells and, if they are set to the region we want, they contribute to
the region's total area.
% This creates some impossible scenarios, though.

constrainAreas(_,_, Length, Length).
constrainAreas(Indexed, Regions, Index, Length) :-
constrainArea(Index, Indexed, Regions,_),
IndexNew #= Index + 1,
constrainAreas(Indexed, Regions, IndexNew, Length).

constrainArea(Index, Indexed, Regions, GottenArea) :-
member([Index, Area_1, _, _], Indexed),
Area #= Area_1 + 1,
getArea(Index, _, Regions, GottenArea),
GottenArea #= Area.

getArea(Index, _, Regions, GottenArea) :-
length(Regions, N),
getAreaRows(0, N, Index, Regions, GottenArea).

getAreaRows(N,N,_,_,0).
getAreaRows(Y,N,Index,Regions,GottenArea) :-
getAreaCols(0,Y,N,Index,Regions,ThisGottenArea),
Y1 #= Y + 1,
getAreaRows(Y1, N, Index, Regions, OtherGottenArea),
GottenArea #= OtherGottenArea + ThisGottenArea.

getAreaCols(N, _, N, _, _, 0).
getAreaCols(X, Y, N, Index, Regions, GottenArea) :-
```

```
getNumberAtCoord(X, Y, Var, Regions),

Var #= Index #<=> B,

X1 #= X + 1,
getAreaCols(X1, Y, N, Index, Regions, ThisGottenArea),
GottenArea #= ThisGottenArea + B.

% This predicate restricts the board so that no region can be disjoint with itself,
ie, it removes scenarios like this one (numbers indicate region index):

%       0 1 0 0 0
%       0 1 1 1 1           Look at region 0. It's disjoint with itself.
%       2 1 3 3 3
%       2 4 4 4 4
%       2 5 5 5 5

%       0 0 0 0 0
%       1 1 1 1 1           Should look like this.
%       2 1 3 3 3
%       2 4 4 4 4
%       2 5 5 5 5

constrainCohesions(_,_, Length, Length).
constrainCohesions(Indexed, Regions, Index, Length) :-
constrainCohesion(Indexed, Regions, Index),
IndexNew #= Index + 1,
constrainCohesions(Indexed, Regions, IndexNew, Length).

constrainCohesion(Indexed, Regions, Index) :-
member([Index, Area_1, X, Y], Indexed),

constrainCohesionUp(Index, Regions, X, Y, AreaUp),
constrainCohesionDown(Index, Regions, X, Y, AreaDown),
constrainCohesionLeft(Index, Regions, X, Y, AreaLeft),
constrainCohesionRight(Index, Regions, X, Y, AreaRight),

OffsettedUp #= AreaUp - 1, OffsettedDown #= AreaDown - 1,
OffsettedLeft #= AreaLeft - 1, OffsettedRight #= AreaRight - 1,

Area_1 #= OffsettedUp + OffsettedDown + OffsettedLeft + OffsettedRight.
```

```
constrainCohesionUp(_, _, _, -1, 0).
constrainCohesionUp(Index, Regions, X, Y, Area) :-
getNumberAtCoord(X, Y, Var, Regions),

Var #= Index,

NewY #= Y - 1,
constrainCohesionUp(Index, Regions, X, NewY, NewArea),
Area #= NewArea + 1.
constrainCohesionUp(Index, Regions, X, Y, Area) :-
getNumberAtCoord(X, Y, Var, Regions),

Var #\= Index,
Area #= 0.

constrainCohesionDown(_, Regions, _, Y, 0) :- length(Regions, Y).
constrainCohesionDown(Index, Regions, X, Y, Area) :-
getNumberAtCoord(X, Y, Var, Regions),

Var #= Index,

NewY #= Y + 1,
constrainCohesionDown(Index, Regions, X, NewY, NewArea),
Area #= NewArea + 1.
constrainCohesionDown(Index, Regions, X, Y, Area) :-
getNumberAtCoord(X, Y, Var, Regions),

Var #\= Index,
Area #= 0.

constrainCohesionLeft(_, _, -1, _, 0).
constrainCohesionLeft(Index, Regions, X, Y, Area) :-
getNumberAtCoord(X, Y, Var, Regions),

Var #= Index,

NewX #= X - 1,
constrainCohesionLeft(Index, Regions, NewX, Y, NewArea),
Area #= NewArea + 1.
constrainCohesionLeft(Index, Regions, X, Y, Area) :-
getNumberAtCoord(X, Y, Var, Regions),

Var #\= Index,
```

```
Area #= 0.

constrainCohesionRight(_, Regions, X, _, 0) :- length(Regions, X).
constrainCohesionRight(Index, Regions, X, Y, Area) :-
getNumberAtCoord(X, Y, Var, Regions),

Var #= Index,

NewX #= X + 1,
constrainCohesionRight(Index, Regions, NewX, Y, NewArea),
Area #= NewArea + 1.
constrainCohesionRight(Index, Regions, X, Y, Area) :-
getNumberAtCoord(X, Y, Var, Regions),

Var #\= Index,
Area #= 0.
```

printer.pl

```
% A predicate to print the element, whether it is a capital, a dash or an arrow.


printElement(Elem-Indexed-_, X, Y,_) :-
member([Elem, Area, CapitalX, CapitalY], Indexed),
CapitalX #= X, CapitalY #= Y,
Area #< 10,
write(' '),
write(Area),
write(' '),
printColumnSeparator.

% ^
printElement(Elem-Indexed-Solution,X,Y,_) :-
member([Elem, _,CapitalX, CapitalY], Indexed),
CapitalX #= X,
Y  #< CapitalY,

isLastOnBranch(Elem-Indexed-Solution,X,Y,0,-1),

write(' ^'),
write(' '),
printColumnSeparator.
```

```
printElement(Elem-Indexed-Solution,X,Y,_) :-
member([Elem, _,CapitalX, CapitalY], Indexed),
CapitalX #= X,
Y  #< CapitalY,

\+ isLastOnBranch(Elem-Indexed-Solution,X,Y,0,-1),

write(' |'),
write(' '),
printColumnSeparator.

% v
printElement(Elem-Indexed-Solution,X,Y,_) :-
member([Elem, _,CapitalX, CapitalY], Indexed),
CapitalX #= X,
Y #> CapitalY,

isLastOnBranch(Elem-Indexed-Solution,X,Y,0,1),

write(' !'),
write(' '),
printColumnSeparator.

printElement(Elem-Indexed-Solution,X,Y,_) :-
member([Elem, _,CapitalX, CapitalY], Indexed),
CapitalX #= X,
Y #> CapitalY,

\+ isLastOnBranch(Elem-Indexed-Solution,X,Y,0,1),

write(' |'),
write(' '),
printColumnSeparator.

% <
printElement(Elem-Indexed-Solution,X,Y,_) :-
member([Elem, _,CapitalX, CapitalY], Indexed),
CapitalY #= Y,
X  #< CapitalX,

isLastOnBranch(Elem-Indexed-Solution,X,Y,-1,0),

write(' '),
```

```
write('<-'),
printColumnSeparator.

printElement(Elem-Indexed-Solution,X,Y,_) :-
member([Elem, _,CapitalX, CapitalY], Indexed),
CapitalY #= Y,
X  #< CapitalX,

\+ isLastOnBranch(Elem-Indexed-Solution,X,Y,-1,0),

write('---'),
printColumnSeparator.

% >
printElement(Elem-Indexed-Solution,X,Y,_) :-
member([Elem, _,CapitalX, CapitalY], Indexed),
CapitalY #= Y,
X  #> CapitalX,

isLastOnBranch(Elem-Indexed-Solution,X,Y,1,0),

write('->'),
write(' '),
printColumnSeparator.

printElement(Elem-Indexed-Solution,X,Y,_) :-
member([Elem, _,CapitalX, CapitalY], Indexed),
CapitalY #= Y,
X  #> CapitalX,

\+ isLastOnBranch(Elem-Indexed-Solution,X,Y,1,0),

write('---'),
printColumnSeparator.

% The predicate to print the solution itself

printSolution(Solution-Indexed) :-
length(Solution, N),
nth0(0, Solution, Row),
write(' '),
printRowHeader(Row), nl,
printSolutionAux(Solution-Indexed, 0, N).
```

```
printSolutionAux(_-_, N, N).
printSolutionAux(Solution-Indexed, I, N) :-
nth0(I, Solution, Row),
printColumnSeparator,
printRow(Row-Indexed-Solution, 0, I, N), nl,
printColumnSeparator, printRowAppearance(Row), nl,
NewI #= I + 1,
printSolutionAux(Solution-Indexed, NewI, N).

printRow(_, N, _, N).
printRow(Row-Indexed-Solution, X, Y, N):-
nth0(X, Row, Elem),
printElement(Elem-Indexed-Solution, X, Y, N),
NewX #= X + 1,
printRow(Row-Indexed-Solution, NewX, Y, N).

printColumnSeparator :- write('|').

printRowHeader([]).
printRowHeader([_| Tail]) :-
        printRowUnderline,
        write(' '),
        printRowHeader(Tail).

printRowAppearance([]).
printRowAppearance([_ | Tail]) :-
printRowUnderline,
printColumnSeparator,
printRowAppearance(Tail).
printRowUnderline :- write('___').


% A predicate to print the unsolved puzzle.

printUnsolvedPuzzle(Puzzle):-
length(Puzzle, N),
nth0(0, Puzzle, Row),
write(' '), printRowHeader(Row),
nl,
printUnsolvedPuzzle_Aux(Puzzle, 0, N).

printUnsolvedPuzzle_Aux(_,N,N).
```

```
printUnsolvedPuzzle_Aux(Puzzle, Y, N) :-
nth0(Y,Puzzle, Row),
printColumnSeparator,
printUnsolvedPuzzleRow(Row, 0, N),
NewY #= Y + 1,
nl,
printColumnSeparator,
printRowAppearance(Row),
nl,
printUnsolvedPuzzle_Aux(Puzzle, NewY, N).

printUnsolvedPuzzleRow(_, N, N).
printUnsolvedPuzzleRow(Row, X, N) :-
nth0(X, Row, Elem),
printUnsolvedElem(Elem),
printColumnSeparator,
NewX #= X + 1,
printUnsolvedPuzzleRow(Row, NewX, N).

printUnsolvedElem(Elem) :-
Elem >= 10,
write(Elem), write(' ').
printUnsolvedElem(Elem) :-
Elem < 0,
write('   ').
printUnsolvedElem(Elem) :-
Elem < 10,
write(' '), write(Elem), write(' ').

% A predicate to convert a list to a matrix of a given side.
% Used to convert the flattened out solution back into an easier-to-print 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).
```

```
% A predicate used to determine whether this cell is the last on this region's branch. Used to check if it should be a dash or an arrow.

% ^
isLastOnBranch(_-_-_,_,Y,0,-1) :-
Y #= 0.
isLastOnBranch(Elem-_-Solution,X,Y,0,-1) :-
Y #\= 0,
Above #= Y - 1,
nth0(Above, Solution, Row), nth0(X, Row, Neighboor),
Neighboor #\= Elem.

% v
isLastOnBranch(_-_-Solution,_,Y,0,1) :-
length(Solution, N),
Y #= N - 1.
isLastOnBranch(Elem-_-Solution,X,Y,0,1) :-
length(Solution, N),
Y #\= N - 1,
Below #= Y + 1,
nth0(Below, Solution, Row), nth0(X, Row, Neighboor),
Neighboor #\= Elem.

% <
isLastOnBranch(_-_-_,X,_,-1,0) :-
X #= 0.
isLastOnBranch(Elem-_-Solution,X,Y,-1,0) :-
X #\= 0,
Left #= X - 1,
nth0(Y, Solution, Row), nth0(Left, Row, Neighboor),
Neighboor #\= Elem.

%>
isLastOnBranch(_-_-Solution,X,_,1,0) :-
length(Solution, N),
X #= N - 1.
isLastOnBranch(Elem-_-Solution,X,Y,1,0) :-
length(Solution,N),
X #\= N - 1,
Right #= X + 1,
nth0(Y, Solution, Row), nth0(Right, Row, Neighboor),
Neighboor #\= Elem.
```