

Resolução de Problema de Decisão usando Programação em Lógica com Restrições

Hitori

Maria João Mira Paulo, Nuno Ramos

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

Resumo O objetivo deste trabalho é a construção de um programa em Programação em Lógica com restrições para a resolução de um problemas de otimização ou decisão combinatória. O problema de decisão escolhido é baseado num puzzle lógico denominado *Hitori*. Este jogo tem como principal objetivo a eliminação de células para que não existam números repetidos na mesma linha e coluna. Através da manipulação de predicados disponibilizados pelo SICStus Prolog, mostramos, neste artigo que foi possível a resolução deste problema de forma eficiente.

1 Introdução

Este projeto, desenvolvido no âmbito da disciplina de Programação em Lógica, consiste na resolução de um problema de otimização ou decisão combinatória através da uso de restrições e com o apoio da biblioteca 'clpfd' presente no SICStus - Prolog. O problema escolhido pelo grupo trata-se de um puzzle chamado *Hitori*.

Este puzzle consiste num jogo de eliminação de números inteiros, jogado num tabuleiro quadrado. O objetivo é eliminar algumas células do tabuleiro de modo a que não haja nenhum número duplicado em nenhuma fila ou coluna. Além disso, os quadrados (números) eliminados não devem tocar-se verticalmente ou horizontalmente, apenas na diagonal; enquanto que todos os quadrados não eliminados devem estar conectados horizontalmente ou verticalmente de forma que seja criada uma única área contígua.

Este artigo explica o problema em questão detalhadamente e além disso demonstra a resolução deste problema em detalhe.

2 Descrição do Problema

O puzzle Hitori é jogado num tabuleiro quadrado em que cada célula contém um número inteiro, como mostra na figura abaixo.

O jogo começa com o tabuleiro completo e deve-se eliminar números tal que não haja nenhum dígito repetido mais do que uma vez em qualquer linha ou coluna. O número eliminado deve ser riscado do tabuleiro.

4	8	1	6	3	2	5	7
3	6	7	2	1	6	5	4
2	3	4	8	2	8	6	1
4	1	6	5	7	7	3	5
7	2	3	1	8	5	1	2
3	5	6	7	3	1	8	4
6	4	2	3	5	4	7	8
8	7	1	4	2	3	5	6

Figura 1. Exemplo de Puzzle Hitori

Além disso, não pode existir nenhum dígito eliminado ligado verticalmente ou horizontalmente a outro dígito eliminado. Dígitos eliminados só se podem conectar diagonalmente.

Finalmente, o puzzle final deve garantir que todos as células não eliminadas devem estar conectadas horizontalmente ou verticalmente de forma que seja criada uma única área contígua.

A solução do puzzle acima encontra-se representada na figura abaixo.

■	8	■	6	3	2	■	7
3	6	7	2	1	■	5	4
■	3	4	■	2	8	6	1
4	1	■	5	7	■	3	■
7	■	3	■	8	5	1	2
■	5	6	7	■	1	8	■
6	■	2	3	5	4	7	8
8	7	1	4	■	3	■	6

Figura 2. Solução Puzzle Hitori

3 Abordagem

O primeiro passo na abordagem foi tentar perceber como modelar o puzzle como um problema de restrições. O grupo empenhou-se em entender as variáveis de decisão a utilizar no predicado de labeling, a forma mais correta de restringir essas variáveis e a maneira mais intuitiva de interagir com o utilizador.

3.1 Variáveis de Decisão

A solução pretendida para este puzzle é o próprio tabuleiro com alguns números eliminados, de forma a que não hajam números repetidos na mesma coluna e linha. Neste sentido, a única variável de decisão (ou variável de domínio) que o

nosso problema necessita para ser resolvido, e a única utilizada no nosso predicado *labeling*, é uma variável chamada *PuzzleSolution*, que se trata de uma lista de listas.

3.2 Restrições impostas à variável *Results*

Em primeiro lugar, na inicialização da variável de decisão foi imposto que em cada célula, o domínio poderia ser ou o valor já existente ou um valor maior que o tamanho do tabuleiro.

Assim, no caso de um jogo cujo tabuleiro tem de dimensões 8 por 8, uma célula tem como domínio ou o próprio valor ou um valor maior ou igual que 9 (tamanho do tabuleiro + 1) e menor que 8*8 apenas como forma de garantir que existem valores diferentes para qualquer célula que tenha que ser eliminada.

```

1 initializeBoard([], [], _).
2 initializeBoard([Line|PuzzleSolution], [LinePuzzle|Puzzle], Size):-
3     initializeLine(Line, LinePuzzle, Size),
4     initializeBoard(PuzzleSolution, Puzzle, Size).
5
6 initializeLine([], [], _).
7 initializeLine([S1|LineSolution], [P1|LinePuzzle], Size):-
8     MaxValue #= Size * Size,
9     N #= Size+1,
10    S1 in (P1..P1) \\/ (N..MaxValue),
11    initializeLine(LineSolution, LinePuzzle, Size).
```

De seguida, foi necessário garantir que não existia nenhum número repetido na mesma linha e coluna. Para isso, foi usado o predicado *all_distinct* linha a linha. Para testar a mesma restrição mas desta vez coluna a coluna, foi necessário inverter a matriz e chamar de novo o predicado *all_distinct*.

```

1 transpose(PuzzleSolution, TransposePuzzleSolution)
2 maplist(all_distinct, PuzzleSolution)
3 maplist(all_distinct, TransposePuzzleSolution)
```

Por fim, asseguramos que nenhuma célula eliminada está conectada horizontalmente ou verticalmente a outra célula eliminada. Para isso, criamos o predicado *checkAdjacentPosition* que para cada linha verifica se alguma célula eliminada está ao lado de outra célula eliminada.

Este predicado é chamado para todo o tabuleiro através de *maplist*. De seguida inverte-se a matriz através do predicado *transpose* e volta-se a chamar o mesmo predicado de novo para todo o tabuleiro.

```

1 checkAdjacentPositions(_,[_]).
2 checkAdjacentPositions(Size,[E1,E2|Line]):-
3     #\ (E1 #> Size #/\ E2 #> Size),
4     checkAdjacentPositions(Size,[E2|Line]).

```

Concluindo, este conjunto de restrições é chamado através do predicado *solvePuzzle* capaz de resolver o puzzle *Puzzle*, retornando a solução em *PuzzleSolution*.

```

1 solvePuzzle(Puzzle,Size,PuzzleSolution):-
2     initializeBoard(PuzzleSolution,Puzzle,Size),
3
4     % inverte o tabuleiro
5     transpose(PuzzleSolution,TransposePuzzleSolution),
6
7     % não permitir elementos diferentes nas linhas e colunas
8     maplist(all_distinct,PuzzleSolution),
9     maplist(all_distinct,TransposePuzzleSolution),
10
11     % não permitir posições adjacentes a preto
12     maplist(checkAdjacentPositions(Size),PuzzleSolution),
13     maplist(checkAdjacentPositions(Size),TransposePuzzleSolution),
14
15     maplist(labeling([]),PuzzleSolution).

```

3.3 Gerador Aleatório do Problema a resolver

Segue-se o método encontrado para a geração de um puzzle aleatório. Inicialmente, gerou-se um tamanho aleatório entre 6 e 10, através do predicado *randomSize*.

De seguida, utilizou-se o predicado *randomBoard*, que é responsável por criar um puzzle, de dimensões *Size*Size*, preenchido por variáveis não instanciadas.

Agora, com um puzzle não instanciado e com as dimensões corretas, através do predicado *randomBoardRestrictions*, instanciámos o puzzle. Este predicado assegura que existem valores em todas as posições do puzzle e que, além disso não existam valores repetidos na mesma linha e coluna. O puzzle gerado é uma solução sem células eliminadas.

```

1 randomBoardRestrictions(Puzzle,Size):-
2     length(Puzzle,Size),

```

```

3   initializeRandomLine(Puzzle,Size),
4   maplist(all_distinct,Puzzle),
5   transpose(Puzzle,TransposePuzzle),
6   maplist(all_distinct,TransposePuzzle),
7   maplist(labeling([]),Puzzle).

```

Com o puzzle solução gerado, é chamado o predicado *fillBoard*. Este predicado é responsável por retornar um puzzle não resolvido. Percorre-se o puzzle, linha a linha, e em cada uma delas altera-se alguns valores para números já existentes. Para isso faz-se um random entre 1 e 4. Se o número gerado pelo random for 1, altera-se o valor desse elemento para um valor já existente, caso contrário, mantém-se o valor. Desta forma, garante-se que cada célula de uma linha tenha probabilidade de 25% de se tornar um elemento duplicado.

```

1   fillBoard([L1|Puzzle],Size,[L2|NewBoard]):-
2       fillLine(L1,Size,L2),
3       fillBoard(Puzzle,Size,NewBoard).
4
5   fillBoard([],_Size,[]).
6
7   fillLine([E1|Line],Size,[E2|NewLine]):-
8       random(1,5,IsBlack),
9       fillElement(E1,E2,Size,IsBlack),
10      fillLine(Line,Size,NewLine).
11
12  fillLine([],_Size,[]).
13
14  fillElement(E1,E2,Size,1):-
15      SizePlusOne is Size+1,
16      random(1,SizePlusOne,Elem),
17      Elem \= E1,
18      E2 = Elem.
19
20  fillElement(E1,E2,Size,1):-
21      fillElement(E1,E2,Size,1).
22
23  fillElement(E1,E2,Size,Index):-
24      E2 = E1.

```

4 Visualização da Solução

O programa permite a resolução de um puzzle predefinido.

```

1 newPuzzle(Puzzle):-
2   Puzzle = [
3       [4,8,1,6,3,2,5,7],
4       [3,6,7,2,1,6,5,4],
5       [2,3,4,8,2,8,6,1],
6       [4,1,6,5,7,7,3,5],
7       [7,2,3,1,8,5,1,2],
8       [3,5,6,7,3,1,8,4],
9       [6,4,2,3,5,4,7,8],
10      [8,7,1,4,2,3,5,6]].
11
12 hitori(Puzzle, PuzzleSolution):-
13     newPuzzle(Puzzle),
14     length(Puzzle,Size),
15     SecondSize is Size+1,
16     display_board(Puzzle,SecondSize),nl,nl,nl,nl,
17     solvePuzzle(Puzzle,8,PuzzleSolution),
18     display_board(PuzzleSolution,SecondSize).

```

O utilizador consegue assim visualizar um modelo de puzzle e a sua resolução.

```

[| ?- hitori(P,S).
|4|8|1|6|3|2|5|7|
|3|6|7|2|1|6|5|4|
|2|3|4|8|2|8|6|1|
|4|1|6|5|7|7|3|5|
|7|2|3|1|8|5|1|2|
|3|5|6|7|3|1|8|4|
|6|4|2|3|5|4|7|8|
|8|7|1|4|2|3|5|6|

| 8|1|6|3|2|5|7|
|3| 7|2|1|6| 4|
| 3|4|8|2| 6|1|
|4|1|6|5| 7|3| 1|
|7| 3|1|8|5| 2|
| 5| 7| 1|8| 1|
|6|4|2|3|5| 7|8|
|8|7| 4| 3| 6|
P = [[4,8,1,6,3,2,5,7],[3,6,7,2,1,6,5,4],[2,3,4,8,2,8,6,1],[4,1,6,5,7,7,3|...],[7,2,3,1,8,5|...],[
3,5,6,7,3|...],[6,4,2,3|...],[8,7,1|...]],
S = [[9,8,1,6,3,2,5,7],[3,9,7,2,1,6,10,4],[10,3,4,8,2,9,6,1],[4,1,6,5,9,7,3|...],[7,10,3,1,8,5|...
],[11,5,9,7,10|...],[6,4,2,3|...],[8,7,10|...]] ?

```

Figura 3. Template de Puzzle com Resolução

Além disso é possível gerar um tabuleiro aleatório.

```
[| ?- generateRandomBoard.
|1|2|3|2|5|6|
|2|1|4|3|6|5|
|2|4|5|6|1|2|
|3|3|6|5|2|1|
|5|1|1|2|6|6|
|5|5|2|2|3|3|
yes
```

Figura 4. Puzzle Gerado Aleatoriamente

Por fim, podemos ainda visualizar um tabuleiro aleatório e a sua resolução, conseguida através do *solver* programado com restrições.

```
[| ?- randomSolver.
|1|2|3|3|5|6|7|8|
|2|1|4|3|6|5|2|7|
|1|4|6|2|7|1|5|3|
|4|3|2|1|8|7|6|5|
|5|6|7|8|1|2|3|4|
|6|5|8|7|2|1|4|3|
|7|8|5|6|3|4|3|2|
|8|7|6|5|4|6|2|1|

|1|2|3| |5|6|7|8|
|2|1|4|3|6|5| |7|
| |4|6|2|7|1|5|3|
|4|3|2|1|8|7|6|5|
|5|6|7|8|1|2|3|4|
|6|5|8|7|2| |4| |
|7|8|5|6|3|4| |2|
|8|7| |5|4| |2|1|
```

Figura 5. Puzzle Gerado Aleatoriamente e respetiva Resolução

Segue-se outro exemplo de um Puzzle Gerado Aleatoriamente e respetiva Resolução.

```

[] ?- randomSolver.
|1|2|3|4|5|6|
|1|1|4|3|6|5|
|3|4|3|6|1|2|
|4|3|3|5|2|1|
|5|6|1|2|4|4|
|6|5|2|1|4|3|

|1|2| |4|5|6|
| |1|4|3|6|5|
|3|4| |6|1|2|
|4| |3|5|2|1|
|5|6|1|2|4| |
|6|5|2|1| |3|
yes

```

Figura 6. Puzzle Gerado Aleatoriamente e respetiva Resolução

5 Conclusões e Considerações Finais

Após a realização deste trabalho conclui-se que a linguagem Prolog, mais especificamente os módulos de resolução de restrições são bastante poderosos permitindo a resolução de uma ampla variedade de questões de decisão e otimização.

Depois de entendido o funcionamento por de trás de variáveis de decisão, formas de restringir o domínio de variáveis, a maneira como o labeling funciona e, por fim, o potencial de todos os predicados que a biblioteca 'clpfd' disponibiliza, tornou-se mais fácil a resolução do problema proposto. O grupo sentiu mais dificuldade quando tentou aplicar a conectividade das células do tabuleiro ao programa. A tentativa encontra-se nos anexos no ficheiro connectivity.pl.

A solução implementada pelo nosso grupo correspondeu às expetativas.

6 Referências

Referências

1. Hitori: <https://pt.wikipedia.org/wiki/Hitori>
2. Slides da disciplina sobre PLR: https://moodle.up.pt/pluginfile.php/55023/mod_resource/content/5/PLR_SICStus.pdf

7 Anexo

```

1  hitori.pl
2
3  :-use_module(library(clpfd)).
4  :-use_module(library(lists)).
5  :-use_module(library(sets)).
6  :- ensure_loaded('utilities.pl').
7  :- ensure_loaded('randomBoard.pl').
8
9  newPuzzle(Puzzle):-
10  Puzzle = [
11      [4,8,1,6,3,2,5,7],
12      [3,6,7,2,1,6,5,4],
13      [2,3,4,8,2,8,6,1],
14      [4,1,6,5,7,7,3,5],
15      [7,2,3,1,8,5,1,2],
16      [3,5,6,7,3,1,8,4],
17      [6,4,2,3,5,4,7,8],
18      [8,7,1,4,2,3,5,6]].
19
20  initializeBoard([],[],_).
21  initializeBoard([Line|PuzzleSolution],[LinePuzzle|Puzzle],Size):-
22      initializeLine(Line,LinePuzzle,Size),
23      initializeBoard(PuzzleSolution,Puzzle,Size).
24
25  initializeLine([],[],_).
26  initializeLine([S1|LineSolution],[P1|LinePuzzle],Size):-
27      MaxValue #= Size * Size,
28      N #= Size+1,
29      S1 in (P1..P1) \\/ (N..MaxValue),
30      initializeLine(LineSolution,LinePuzzle,Size).
31
32  checkAdjacentPositions(_,[_]).
33  checkAdjacentPositions(Size,[E1,E2|Line]):-
34      #\ (E1 #> Size #/\ E2 #> Size),
35      checkAdjacentPositions(Size,[E2|Line]).
36
37  solvePuzzle(Puzzle,Size,PuzzleSolution):-
38      initializeBoard(PuzzleSolution,Puzzle,Size),
39      transpose(PuzzleSolution,TransposePuzzleSolution),

```

```

40  maplist(all_distinct,PuzzleSolution),
41  maplist(all_distinct,TransposePuzzleSolution),
42  maplist(checkAdjacentPositions(Size),PuzzleSolution),
43  maplist(checkAdjacentPositions(Size),TransposePuzzleSolution),
44  maplist(labeling([]),PuzzleSolution).
45
46  randomSolver:-
47      randomSize(Size),
48      randomBoard(Puzzle2,Size),
49      randomBoardRestrictions(Puzzle,Size),
50      fillBoard(Puzzle,Size,Puzzle2),
51      SizePlusOne is Size+1,
52      display_board(Puzzle2,SizePlusOne),nl,nl,nl,nl,
53      solvePuzzle(Puzzle2,Size,Solution),
54      display_board(Solution,SizePlusOne).
55
56  generateRandomBoard:-
57      randomSize(Size),
58      randomBoard(Puzzle2,Size),
59      randomBoardRestrictions(Puzzle,Size),
60      fillBoard(Puzzle,Size,Puzzle2),
61      SizePlusOne is Size+1,
62      display_board(Puzzle2,SizePlusOne).
63
64  hitori(Puzzle, PuzzleSolution):-
65      newPuzzle(Puzzle),
66      length(Puzzle,Size),
67      SecondSize is Size+1,
68      display_board(Puzzle,SecondSize),nl,nl,nl,nl,
69      solvePuzzle(Puzzle,8,PuzzleSolution),
70      display_board(PuzzleSolution,SecondSize).

```

```
1  random.pl
2
3  :- use_module(library(random)).
4
5  randomBoard(Board,Size):-
6      length(Board,Size),
7      randomB(Board,Size).
8
9  randomB([],_).
10 randomB([B1|Board],Size):-
11     randomB(Board,Size),
12     length(B1,Size).
13
14 randomSize(Size):-
15     random(6,10,Size).
16
17 fillBoard([L1|Puzzle],Size,[L2|NewBoard]):-
18     fillLine(L1,Size,L2),
19     fillBoard(Puzzle,Size,NewBoard).
20
21 fillBoard([],_Size,[]).
22
23 fillLine([E1|Line],Size,[E2|NewLine]):-
24     random(1,5,IsBlack),
25     fillElement(E1,E2,Size,IsBlack),
26     fillLine(Line,Size,NewLine).
27
28 fillLine([],_Size,[]).
29
30 fillElement(E1,E2,Size,1):-
31     SizePlusOne is Size+1,
32     random(1,SizePlusOne,Elem),
33     Elem \= E1,
34     E2 = Elem.
35
36 fillElement(E1,E2,Size,1):-
37     fillElement(E1,E2,Size,1).
38
39 fillElement(E1,E2,Size,Index):-
40     E2 = E1.
```

```
41
42 initializeRandomLine([],_NCol).
43
44 initializeRandomLine([Line|Board],NCol):-
45     initializeRandomLine(Board,NCol),
46     length(Line,NCol),
47     domain(Line,1,NCol).
48
49 randomBoardRestrictions(Puzzle,Size):-
50     length(Puzzle,Size),
51     initializeRandomLine(Puzzle,Size),
52     maplist(all_distinct,Puzzle),
53     transpose(Puzzle,TransposePuzzle),
54     maplist(all_distinct,TransposePuzzle),
55     maplist(labeling([]),Puzzle).
```

```
1  utilities.pl
2
3  display_board([L1|LS], MaxValue):-
4      write('|'),
5      display_line(L1,MaxValue), nl,
6      display_board(LS,MaxValue).
7
8  display_board([],_MaxValue).
9
10 display_line([E1|ES],MaxValue):-
11     E1 < MaxValue,
12     write(E1),
13     write('|'),
14     display_line(ES,MaxValue).
15
16 display_line([_E1|ES],MaxValue):-
17     write(' '),
18     write('|'),
19     display_line(ES,MaxValue).
20
21 display_line([],_MaxValue).
```

```

1 Tentativa de Conectividade - connectivity.pl
2
3 checkConnectivity(_,[_,_],[_,_]).
4 checkConnectivity(Size,[S1,S2,S3|Solution],[TS1,TS2,TS3|TransposeSolution]):-
5     #\ (S1 #> Size #/\ S3 #> Size #/\ TS1 #> Size #/\ TS3 #> Size),
6     checkConnectivity(Size,[S2,S3|Solution],[TS2,TS3|TransposeSolution]).
7
8
9 checkConnectivityCorners(Size,[S1|Solution],[TS1|TransposeSolution]):-
10    % Line = 1,
11    BeforeLast #= Size - 1.
12    element(2,S1,E1),
13    element(2,TS1,TE1),
14    #\ (E1 #> Size #/\ TE1 #> Size).
15    element(2,S1,E1),
16    element(2,TS1,TE1),
17    #\ (E1 #> Size #/\ TE1#> Size)
18    NextLine is Line+1,
19    NextLine #=< Size.
20
21 checkBordersConnectivity(Size,[S1|Solution],[TS1|TransposeSolution]):-
22    BeforeLast #= Size - 1,
23    getLine(2,1,[S1|Solution],S2),
24    compareLines(S1,S2,Size),
25    getLine(Size,1,[S1|Solution],SN),
26    getLine(BeforeLast,1,[S1|Solution],SBL),
27    compareLines(SN,SBL,Size),
28    getLine(2,1,[TS1|TransposeSolution],TS2),
29    compareLines(TS1,TS2,Size),
30    getLine(Size,1,[TS1|TransposeSolution],TSN),
31    getLine(BeforeLast,1,[TS1|TransposeSolution],TBL),
32    compareLines(TSN,TBL,Size).
33
34
35 getLine(PositionWanted,Index,[S1|_],SN):-
36     PositionWanted = Index,
37     S1 = SN.
38
39 getLine(PositionWanted,Index,[_|Solution],SN):-
40     NextIndex is Index+1,

```

```

41     getLine(PositionWanted,NextIndex,Solution,SN).
42
43 compareLines([_,_],[_,_],_Size).
44 compareLines([E1,E2,E3|ES],[_R1,R2,R3|RS],Size):-
45     #\ ( E1 #> Size #/\ E2 #=< Size #/\ E3 #> Size #/\ R2 #> Size),
46     compareLines([E2,E3|ES],[R2,R3|RS],Size).
47
48 flattenList([],[]).
49 flattenList([L1|Ls], Lf):- is_list(L1), flattenList(L1, L2),
50     append(L2, Ld, Lf),
51     flattenList(Ls, Ld).
52 flattenList([L1|Ls], [L1|Lf]):- \+is_list(L1), flattenList(Ls, Lf).

```
