Resolução do problema de decisão *Fence* usando programação em lógica com restrições

Bruno Pinto[12] e João Mendes[13]

[1] FEUP-PLOG, Turma 3MIEIC06, Grupo Fence_5
[2] up201502960@fe.up.pt
[3] up201505439@fe.up.pt

Resumo O presente artigo serve de complemento ao segundo projeto da Unidade Curricular de Programação em Lógica. O projeto consistiu em escrever um programa em *Prolog*, capaz de resolver qualquer tabuleiro do puzzle *Fence*. Este é um problema de decisão resolvido utilizando restrições que no seu conjunto nos levam a uma solução possível. O resultado final aponta para a utilidade da biblioteca *clpfd* e para a eficácia do *Prolog* em facilitar a resolução de problemas de alta complexidade, apesar de ser reconhecido que a solução obtida não é ótima.

Keywords: feup, prolog, sicstus, fence

1 Introdução

O objetivo deste projeto era implementar a resolução de um problema de decisão ou otimização em linguagem *Prolog*, recorrendo a restrições.

Da lista dos problemas propostos, o nosso grupo optou pelo puzzle *Fence*, que é um problema de decisão.

No restante deste artigo, descreve-se: o puzzle *Fence*; a abordagem de implementação, bem como a análise da mesma; a visualização da solução, incluindo a sua explicação; os resultados obtidos, através de estatísticas de resolução de puzzles com diferentes complexidades. Por último, apresentamos a conclusão a que chegamos com o desenvolvimento deste programa.

2 Descrição do Problema

Slitherlink é jogado em uma rede retangular de pontos. Alguns dos quadrados formados pelos pontos têm números dentro deles. O objetivo é conectar pontos adjacentes horizontal e verticalmente para que as linhas formem um loop simples sem extremidades soltas. Além disso, o número dentro de um quadrado representa quantos dos seus quatro lados são segmentos no circuito.

O puzzle *Fence* consiste numa estrutura retangular de pontos, onde há células, ou quadrados de pontos, preenchidas com uma quantidade não-fixa de números inteiros, que variam no intervalo fechado de 0 a 3.

O objetivo do puzzle é desenhar um único ciclo fechado que não se toca ou se cruza a si próprio. Isto obriga a que uma célula vazia não possa ter 4 *fences*.

Os inteiros representam a quantidade de *fences* que estão nas bordas da respetiva célula. As células vazias podem ter um número qualquer de *fences*, entre 0 e 3.

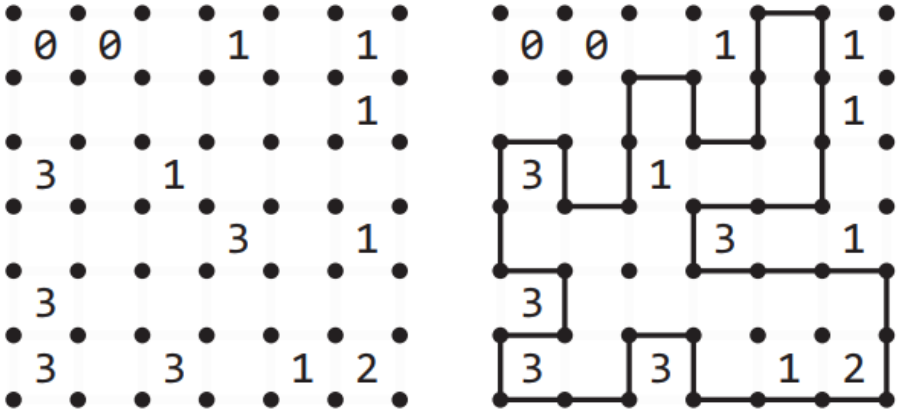

Figura 1. Exemplo de um tabuleiro de Fence com dimensões 6x6 com resolução; 14 células com dígitos (22%)

3 Representação do Problema

De forma a representarmos os factos que decorrem de uma instância de um puzzle *Fence*, usamos os seguintes predicados:

- **cell(+Digit, +Line, +Column)** - indica o número (*Digit*) que se encontra na célula, cujo ponto do canto superior esquerdo tem coordenadas(*Line*, *Column*).
- **dimensions(+horPoints, +verPoints)** - indicam a dimensão da grelha de pontos do problema, dada por *horPoints*x*verPoints*.

Esta representação permite uma mais fácil abstração do problema para consequente resolução e visualização.

4 Abordagem

Este problema foi modelado como um **P**roblema de **S**atisfação de **R**estrições e seguem-se as descrições das variáveis de decisão, das restrições, da função de avaliação e da estratégia de pesquisa utilizadas na resolução deste problema de decisão.

4.1 Variáveis de Decisão

As variáveis de decisão do nosso problema são os *fences*, isto é, as *M*x*(N-1)* linhas e as *(M-1)*x*N* colunas (em que *M* é o número de linhas de células e *N* é o número de colunas de células) passíveis de serem preenchidas. O seu domínio discreto limita-se aos valores 0, representando uma linha ou aresta não preenchida, e 1, representado uma linha ou aresta preenchida.

Estes valores binários são utilizados tanto para a resolução do problema como para a sua visualização.

4.2 Restrições

Para a resolução do puzzle foram implementadas duas restrições, a partir dos seguintes predicados:

- **cellRestriction(+Database, Lines, Columns)** - obriga a que a quantidade de arestas que circundam uma casa preenchida com número, seja igual ao número em questão;
- **pointRestriction(Lines, Columns)** - obriga a que o número de *fences* a sair de um dado ponto seja 0, ou 2, e, por conseguinte, a que o ciclo obtido seja fechado. Com isto evitamos também que o ciclo se toque ou cruze;

Para além destas restrições, implementadas usando **P**rogramação em **L**ógica com **R**estrições, foi ainda adicionada um predicado que verifica se a solução obtida tem apenas um ciclo, causando *backtracking* no caso contrário: **oneLoopVerification(+Lines, +Columns, +NoLines, +NoColumns)**. Admite-se que esta solução obtida por procura por força bruta não é a mais eficiente.

4.3 Estratégia de Pesquisa

A escolha da estratégia de pesquisa recaiu sobre o *labeling* com as opções *default*, uma vez que achamos que nenhuma das opções se adequava ao problema em questão.

5 Visualização da Solução

De forma a visualizar a solução em modo de texto, de uma forma *user friendly*, foram utilizados os seguintes predicados:

- **printing(+Lines, +Columns)** - Processa as listas de listas *Lines* e *Columns*, chamando predicados que mostram o seu conteúdo após substituição dos valores binários - 1 e 0 - pela sua representação gráfica;
- **replace(+Replaced, +Replacement, +List, -NewList)** - Usado para substituir os valores binários - 1 e 0 - pela sua representação gráfica;
- **printSingleLine(+Line)** - Faz a impressão de uma linha da lista de linhas *Lines*;

– **printSingleColumn(+Column)** - Faz a impressão de uma linha da lista de colunas *Columns*, intercalando com a impressão do dígito que uma dada célula contenha, ou de um espaço ' ', caso esteja vazia.

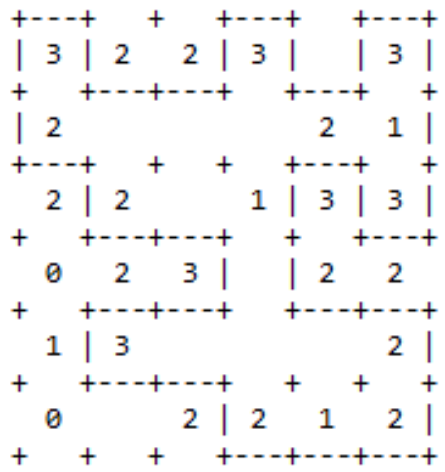

Figura 2. Solução de um problema, como é apresentada pela programa

```
Lines = [[0,0,1,0],[0,1,0,1],[1,0,0,0],[0,1,0,1],[1,0,1,0]]
Columns = [[0,0,1,1,0],[0,1,0,0,1],[1,0,0,0,1],[1,1,1,1,0]]
```

Figura 3. Solução de um problema, como é representado em Prolog

6 Resultados

É possível analisar os resultados estatísticos obtidos para puzzles com diferentes tamanhos da grelha de pontos e diferentes quantidades de células preenchidas. Segue-se uma tabela que contem informações úteis para essa mesma análise, que são:

– **Reatamentos** - número de vezes que uma restrição foi retomada;
– **Implicações** - número de vezes que um (des)entendimento foi detectado por uma restrição;
– **Podas** - número de vezes que um domínio foi podado;
– ***Backtracks*** - número de vezes que uma contradição foi encontrada por um domínio extinto;
– ***Restrições*** - número de restrições criadas.

Puzzle	Reatamentos	Implicações	Podas	*Backtracks*	Restrições	Tempo (ms)
5 x 5 (34%)	808	321	737	4	285	40
5 x 5 (44%)	753	279	690	2	293	47
6 x 6 (40%)	6749	3778	5671	56	422	1172
6 x 6 (48%)	1002	353	953	1	431	36
7 x 7 (44%)	2730	1281	2390	18	591	40
7 x 7 (72%)	1413	519	1364	1	634	44

Tabela 1. Resultados obtidos para diferentes tamanhos de puzzle com diferentes quantidades de células preenchidas

Podemos assim tirar algumas conclusões desta análise:

- A quantidade de reatamentos, implicações, podas e restrições tende a acompanhar o aumento das dimensões do puzzle;
- O número de backtracks é variável, parecendo não ter qualquer correlação com as dimensões do puzzle, mas com a quantidade de células preenchidas no mesmo;
- O tempo de execução aparenta ser semelhante para problemas de complexidade de resolução semelhante.

7 Conclusões

Deste projeto pudemos concluir que a linguagem *Prolog* mostrou-se uma ferramenta bastante útil na resolução de Problemas de Satisfação de Restrições. A biblioteca disponibilizada (cplfd) é uma mais valia, que nos trouxe diferentes formas de atacar o problema e a implementação foi menos complexa do que seria caso estivéssemos restringidos às mesmas ferramentas do que no anterior projeto.

Houve alguma dificuldade numa primeira fase, para a estrutura de dados a utilizar, mas olhando para trás achamos que seguimos pelo melhor caminho com uma estrutura linear que se provou, aliada ao *Prolog*, rápida na resolução de problemas.

Na visualização, foi um pouco moroso até ficar apelativo ao utilizador, pois o *Prolog* não oferece boas ferramentas nesse aspeto. No entanto, tal é compreensível porque foi uma linguagem construída a pensar em lógica, e, por isso, bastante favorável a esta Unidade Curricular.

Estes problemas levaram-nos um pouco a pensar em autómatos finitos, onde as restrições servem como estados e as suas verificações mudanças de estado afetas ao domínio. Podemos então ver as soluções como estados finais, verificando idealmente todas as restrições. No entanto, acabamos por não seguir essa hipótese.

O *Fence* mostrou-nos que jogos simples e com um pequeno número de regras podem ser uma dor de cabeça na sua implementação, e que a lógica por detrás

dos mesmos pode ser bastante complexa. De forma subjetiva, o puzzle não é tão fácil de resolver como seria de esperar e o *Prolog* foi uma ajuda fulcral para o entendimento não só do jogo, mas, generalizando, dos problemas PSR.

Referências

1. Fence rules,
   `http://nikoli.co.jp/en/puzzles/slitherlink.html`
   `https://en.wikipedia.org/wiki/Slitherlink`
2. Fence gameplay,
   `https://www.kakuro-online.com/slitherlink/`
3. SWI-Prolog,
   `http://www.swi-prolog.org/`
4. SICStus Prolog,
   `https://sicstus.sics.se/`

Anexo

Código fonte

puzzle.pl

```
1  :-include('gamePrinting.pl').
2  :-include('restrictions.pl').
3  :-include('verification.pl').
4  :-use_module(library(clpfd)).
5  :-use_module(library(lists)).
6  :-use_module(library(statistics)).
7
8  puzzle  :-
9    dimensions(NoLines, NoColumns),
10   length(Lines, NoLines),
11   LengthSubLines is NoColumns - 1,
12   resizeSubs(Lines, LengthSubLines),
13   defineDomain(Lines),
14   LengthColumns is NoLines - 1,
15   length(Columns, LengthColumns),
16   resizeSubs(Columns, NoColumns),
17   defineDomain(Columns),
18   findall([Digit, Line, Column], cell(Digit, Line, Column),
     ↪  Database),
19   pointRestriction(Lines, Columns),
20   cellRestriction(Database, Lines, Columns),
21   !,
22   labelingVars(Lines),
23   labelingVars(Columns),
```

```
24    oneLoopVerification(Lines, Columns, NoLines, NoColumns),
25    printing(Lines, Columns).
26
27  resizeSubs([Elem], Length) :-
28    length(Elem, Length).
29
30  resizeSubs([Head|Tail], Length) :-
31    length(Head, Length),
32    resizeSubs(Tail, Length).
33
34  defineDomain([]).
35  defineDomain([Head|Tail]) :-
36    domain(Head, 0, 1),
37    defineDomain(Tail).
38
39  labelingVars([]).
40  labelingVars([Head|Tail]) :-
41    labeling([], Head),
42    labelingVars(Tail).
```

restrictions.pl

```
1   cellRestriction([], _, _).
2
3   cellRestriction([[Digit, Line, Column] | Tail], Lines, Columns) :-
4       nth1(Line, Lines, TopLine),
5       element(Column, TopLine, Top),
6       NewLine is Line + 1,
7       nth1(NewLine, Lines, BottomLine),
8       element(Column, BottomLine, Bottom),
9       nth1(Line, Columns, Col),
10      element(Column, Col, LeftCol),
11      NewColumn is Column + 1,
12      element(NewColumn, Col, RightCol),
13      sum([Top, Bottom, LeftCol, RightCol], #=, Digit),
14      cellRestriction(Tail, Lines, Columns).
15
16  pointRestriction(Lines, Columns) :-
17   pointRestriction(Lines, Columns, 1, 1).
18
19  pointRestriction(_,_,CurrLine,CurrColumn) :-
20    dimensions(NoLines, NoColumns),
21    OverLines is NoLines + 1,
22    OverColumns is NoColumns + 1,
23    CurrLine = OverLines,
24    CurrColumn = OverColumns.
```

```
25
26  pointRestriction(Lines, Columns, I, J):-
27    dimensions(NoLines, NoColumns),
28    nth1(I, Lines, Line),
29    (
30      J < NoColumns,
31      element(J, Line, RightLine),
32      A=RightLine
33    ;
34      A = 0,
35      true
36    ),
37    (
38      J \= 1, OldJ is J - 1,
39      element(OldJ, Line, LeftLine),
40      B=LeftLine
41      ;
42      B = 0,
43      true
44    ),
45    (
46      I < NoLines,
47      nth1(I, Columns, Down),
48      element(J, Down, DownCol),
49      C=DownCol
50      ;
51      C = 0,
52      true
53    ),
54    (
55      I \= 1, OldI is I - 1,
56      nth1(OldI, Columns, Up),
57      element(J, Up, UpCol),
58      D = UpCol
59      ;
60      D = 0,
61      true
62    ),
63    (
64      J < NoColumns,
65      NewI is I,
66      NewJ is J + 1
67      ;
68      J = NoColumns,
69      I = NoLines,
```

```
    NewJ is NoColumns + 1,
    NewI is NoLines + 1
    ;
    J = NoColumns,
    I \= NoLines,
    NewJ is 1,
    NewI is I+1
  ),
  (((A+B+C+D) #= 2) #\/ ((A+B+C+D) #= 0)) #<=> Bool,
  Bool #= 1,
  pointRestriction(Lines, Columns, NewI, NewJ).
```

verification.pl

```
oneLoopVerification(Lines, Columns, NoLines, NoColumns) :-
  findFirstLine(Lines, Line, Column),
  findLoop(Lines, Columns, Line, Column, Length, NoLines,
  ↪  NoColumns),
  countOnes(Lines, FilledLines),
  countOnes(Columns, FilledColumns),
  Filled is FilledLines + FilledColumns,
  Filled = Length.

indexOf([Element|_], Element, 1):- !.
indexOf([_|Tail], Element, Index):-
  indexOf(Tail, Element, Index1),
  !,
  Index is Index1+1.

countOnes(Matrix, Ones) :-
  countOnes(Matrix, 0, Ones).

countOnes([], Ones, Ones).

countOnes([Head|Tail], Accumulator, Ones) :-
  sumlist(Head, Sum),
  NewAccumulator is Accumulator + Sum,
  countOnes(Tail, NewAccumulator, Ones).

findFirstLine(Lines, Line, Column) :-
  findFirstLine(Lines, 1, Line, Column).

findFirstLine([Head|Tail], ProcLine, Line, Column) :-
  (
    indexOf(Head, 1, Index),
    Line is ProcLine,
```

```
32      Column is Index
33    ;
34      NewLine is ProcLine + 1,
35      findFirstLine(Tail, NewLine, Line, Column)
36    ).
37
38  findLoop(Lines, Columns, Line, Column, Length, NoLines, NoColumns)
    ↪  :-
39    findLoop(Lines, Columns, Line, Column, [Line-Column], List,
      ↪  NoLines, NoColumns),
40    length(List, Length).
41
42  finish(Accumulator, Accumulator).
43
44  findLoop(Lines, Columns, Line, Column, Accumulator, List, NoLines,
    ↪  NoColumns) :-
45    (
46      Line = 1
47      ;
48      OldLine is Line - 1
49    ),
50    (
51      Column = 1
52      ;
53      OldColumn is Column - 1
54    ),
55    nth1(Line, Lines, CurrLine),
56    (
57      Line = NoLines
58      ;
59      nth1(Line, Columns, Bottom)
60    ),
61    (
62      Line = 1
63      ;
64      nth1(OldLine, Columns, Top)
65    ),
66    (
67      (
68        Column \= NoColumns,
69        nth1(Column, CurrLine, RightLine),
70        RightLine = 1,
71        NewColumn is Column + 1,
72        \+member(Line-NewColumn, Accumulator),
73        append([Line-NewColumn], Accumulator, NewAccumulator),
```

```
74        findLoop(Lines, Columns, Line, NewColumn, NewAccumulator,
          ↪  List, NoLines, NoColumns)
75        ;
76        Column \= 1,
77        nth1(OldColumn, CurrLine, LeftLine),
78        LeftLine = 1,
79        \+member(Line-OldColumn, Accumulator),
80        append([Line-OldColumn], Accumulator, NewAccumulator),
81        findLoop(Lines, Columns, Line, OldColumn, NewAccumulator,
          ↪  List, NoLines, NoColumns)
82        ;
83        Line \= NoLines,
84        nth1(Column, Bottom, BottomCol),
85        BottomCol = 1,
86        NewLine is Line + 1,
87        \+member(NewLine-Column, Accumulator),
88        append([NewLine-Column], Accumulator, NewAccumulator),
89        findLoop(Lines, Columns, NewLine, Column, NewAccumulator,
          ↪  List, NoLines, NoColumns)
90        ;
91        Line \= 1,
92        nth1(Column, Top, TopCol),
93        TopCol = 1,
94        \+member(OldLine-Column, Accumulator),
95        append([OldLine-Column], Accumulator, NewAccumulator),
96        findLoop(Lines, Columns, OldLine, Column, NewAccumulator,
          ↪  List, NoLines, NoColumns)
97      )
98    ;
99      finish(Accumulator, List)
100   ).
```

gamePrinting.pl

```
1   printing(Lines, Columns) :-
2           printing(Lines, Columns, 1).
3
4   printing(Lines, _, LineNo):-
5           dimensions(NoLines, _),
6           LineNo = NoLines,
7           nth1(LineNo, Lines, Line),
8           replace(1, '---', Line, NewLine),
9           replace(0, '   ', NewLine, ParsedLine),
10          printSingleLine(ParsedLine).
11
12  printing(Lines, Columns, I):-
```

```
13          nth1(I, Lines, Line),
14          replace(1, '---', Line, NewLine),
15          replace(0, '   ', NewLine, ParsedLine),
16          printSingleLine(ParsedLine),
17          nth1(I, Columns, Column),
18          replace(1, '|', Column, NewColumn),
19          replace(0, ' ', NewColumn, ParsedColumn),
20          printSingleColumn(ParsedColumn, I),
21          NewI is I + 1,
22          printing(Lines, Columns, NewI).
23
24  replace(_, _, [], []).
25  replace(Replaced, Replacement, [Replaced|Tail],
    ↪  [Replacement|Tail2]) :-
26          replace(Replaced, Replacement, Tail, Tail2).
27  replace(Replaced, Replacement, [Head|Tail], [Head|Tail2]) :-
28          Head \= Replaced,
29          replace(Replaced, Replacement, Tail, Tail2).
30
31  printSingleLine([Cell]):-
32          write('+'),
33          write(Cell),
34          write('+'),
35          nl.
36  printSingleLine([Cell|More]):-
37          write('+'),
38          write(Cell),
39          printSingleLine(More).
40
41  printSingleColumn(ParsedColumn, I) :-
42          printSingleColumn(ParsedColumn, I, 1).
43
44  printSingleColumn([Cell], _, _):-
45          write(Cell),
46          nl.
47  printSingleColumn([Cell|More], I, J):-
48          write(Cell),
49          (
50                  cell(Digit, I, J),
51                  write(' '),
52                  write(Digit),
53                  write(' ')
54                  ;
55                  write('   ')
56          ),
```

```
57          NewJ is J + 1,
58          printSingleColumn(More, I, NewJ).
```

game5x5.pl

```
1  cell(1,1,3).
2  cell(0,1,4).
3  cell(2,2,4).
4  cell(1,3,1).
5  cell(3,3,2).
6  cell(1,4,4).
7
8  dimensions(5,5).
```

game5x5_2.pl

```
1  cell(3,1,2).
2  cell(1,2,1).
3  cell(0,2,3).
4  cell(3,2,4).
5  cell(2,3,3).
6  cell(3,3,4).
7  cell(2,4,2).
8  cell(2,4,3).
9
10 dimensions(5,5).
```

game6x6.pl

```
1  cell(2,1,2).
2  cell(2,1,3).
3  cell(3,1,5).
4  cell(2,2,2).
5  cell(3,3,1).
6  cell(2,3,2).
7  cell(1,4,1).
8  cell(0,4,4).
9  cell(3,4,5).
10 cell(2,5,2).
11 cell(2,5,3).
12 cell(3,5,4).
13
14 dimensions(6,6).
```

game6x6_2.pl

```
cell(3,1,1).
cell(1,2,2).
cell(1,2,3).
cell(2,2,5).
cell(0,3,1).
cell(1,3,5).
cell(2,4,1).
cell(2,4,3).
cell(1,4,4).
cell(3,5,5).

dimensions(6,6).
```

game7x7.pl

```
cell(3,1,1).
cell(2,2,1).
cell(2,3,1).
cell(0,4,1).
cell(1,5,1).
cell(0,6,1).
cell(2,1,2).
cell(2,3,2).
cell(2,4,2).
cell(3,5,2).
cell(2,1,3).
cell(3,4,3).
cell(2,6,3).
cell(3,1,4).
cell(1,3,4).
cell(2,6,4).
cell(2,2,5).
cell(3,3,5).
cell(2,4,5).
cell(1,6,5).
cell(3,1,6).
cell(1,2,6).
cell(3,3,6).
cell(2,4,6).
cell(2,5,6).
cell(2,6,6).

dimensions(7,7).
```

game7x7_2.pl

```
cell(1,1,1).
cell(3,1,3).
cell(1,1,4).
cell(3,1,6).
cell(3,2,4).
cell(3,3,2).
cell(2,3,4).
cell(1,3,5).
cell(0,4,2).
cell(2,4,3).
cell(1,4,5).
cell(2,5,3).
cell(2,6,1).
cell(2,6,3).
cell(3,6,4).
cell(3,6,6).

dimensions(7,7).
```