

# Resolução de Problema de Decisão usando Programação em Lógica com Restrições Skyscrapers

José Pedro Borges, Miguel Mano Fernandes

Faculdade de Engenharia da Universidade do Porto, R. Dr. Roberto Frias, 4200-465  
Porto, Portugal,  
candidato@fe.up.pt,  
WWW home page: fe.up.pt,  
FEUP-PLOG, Turma 3MIEIC01, Grupo Skyscrapers\_3

**Resumo** O presente relatório serve como complemento para o trabalho desenvolvido pelo nosso grupo acerca do desenvolvimento de um programa que resolvesse o puzzle de decisão combinatória Skyscrapers utilizando Programação em Lógica com Restrições, escrito em Prolog. Este projeto foi realizado no âmbito da unidade curricular **Programação em Lógica**.

**Keywords:** prolog, cplfd, restrições, skyscrapers

## 1 Introdução

O nosso grupo decidiu implementar o puzzle Skyscrapers de entre todas as opções apresentadas porque nos pareceu um puzzle diferente daqueles a que estamos geralmente habituados e porque achamos que o nível de dificuldade estava na medida certa, sendo algo desafiante sem ser excessivamente árduo.

O puzzle consiste em preencher um tabuleiro inicialmente vazio com números, que representam altura de edifícios, de forma a que nenhum número seja repetido em cada linha e coluna e que se consigam ver tantos edifícios como o número escrito fora do tabuleiro. Edifícios mais altos tapam a vista dos mais baixos.

Este relatório descreve detalhadamente o puzzle referido anteriormente assim como todos os passos tomados pelo grupo na realização do projeto seguidos pela sua explicação e os resultados e estatísticas da resolução de tabuleiros de diversos tamanhos através do nosso programa.

## 2 Descrição do problema

O problema de decisão selecionado foi a implementação de um algoritmo que soluciona automaticamente um puzzle Skyscrapers, seja ele manualmente introduzido ou gerado aleatoriamente.

Em resumo, a grelha é inicializada em branco, devendo ser preenchida na sua íntegra. Para tal, o jogador deverá recorrer às pistas presentes no exterior, assim como as táticas de resolução de puzzles do estilo Sudoku, que gozam da propriedade de unicidade de valores por linha e coluna.

O puzzle em causa declara uma série de regras cruciais para a resolução do mesmo. São elas:

- Cada linha e coluna contém números desde o 1 até ao comprimento da grelha, apenas surgindo uma única vez. Cada número representa a altura do edifício.
- As pistas no exterior do tabuleiro informam o número de arranha-céus visíveis dessa mesma perspetiva.
- Um arranha-céus mais curto não é visível por detrás de um maior.

Resumindo, o objetivo do algoritmo é preencher cada célula com a altura do arranha-céus, de tal modo que, aliado aos valores das restantes células, o número de edifícios visíveis corresponda ao valor da pista da coluna/linha.

## 3 Abordagem

Segue-se uma descrição técnica da abordagem utilizada.

### 3.1 Variáveis de decisão

Para resolução do puzzle em questão, foi criada uma matriz com **N** listas de tamanho **N**, preservando a sua forma quadrangular.

Cada célula será preenchida com um inteiro, que vai desde 1 até **N**, representativo da altura de um edifício.

Foi necessário iterar e transpor a matriz para assegurar que o domínio 1..**N** é aplicado a todas as linhas e colunas.

---

```
1 generate_empty_board(Size, Matrix) :- % Gera matriz vazia de tamanho Size.
2   bagof(R, Y ^ (between(1, Size, Y), length(R, Size)), Matrix).
```

---

### 3.2 Restrições

Rapidamente tornou-se óbvio que duas principais restrições seriam utilizadas para assegurar uma solução eficiente do puzzle:

- Todos os elementos de cada linha e coluna são distintos. Tal é conseguido através da chamada **all\_distinct** a cada parcela da matriz.

---

```
1 maplist(all_distinct, Board), % Elementos das linhas são únicos.
2 transpose(Board, InvertedBoard), % Inverte a matriz.
3 maplist(all_distinct, InvertedBoard), % Elementos das colunas são únicos.
```

---

- Garantir que a quantidade de edifícios vistos da perspectiva da pista no exterior da grelha corresponde ao valor da pista.

Para este último ponto, produziu-se um predicado **get\_seen\_buildings** que recebe uma lista, unificando a variável **Result** ao número de arranha-céus efetivamente vistos dessa posição.

---

```
1 get_seen_buildings([], 0).
2 get_seen_buildings([H|T], Result) :-
3   get_seen_buildings(T, TokenResult),
4   maximum(Max, [H|T]),
5   H #= Max #<=> Seen,
6   Result #= TokenResult + Seen.
```

---

A variável obtida é posteriormente forçada a ser idêntica ao valor da pista associada. **CH1** e **CH2** referem-se ao par de pistas esquerda/direita ou cima/-baixo, enquanto que **BH** corresponde à coluna/linha a ser processada.

Deste modo, é forçado que o número de edifícios vistos de um lado ou de outro da linha corresponda ao número da pista. Daí a inicial reversão da lista, pois **o número de arranha-céus visíveis de uma perspetiva é diferente da simétrica**.

---

```
1 reverse(BH, BH_Rev),
2 CH1 \= 0, get_seen_buildings(BH_Rev, Seen_Rev), Seen_Rev #= CH1,
3 CH2 \= 0, get_seen_buildings(BH, Seen), Seen #= CH2,
```

---

Para a geração aleatória de tabuleiros, em vez de, talvez, a abordagem mais intuitiva, procedeu-se, primeiramente, ao total preenchimento da tabela, aplicando a restrição **all\_distinct** às linhas e colunas. Posteriormente, as pistas associadas a este tabuleiro são extraídas e armazenadas em memória.

Deste modo, é garantida, pelo menos, uma possível solução para o problema.

### 3.3 Função de avaliação

O solução obtida é sempre válida e a melhor, visto o desafio ser um puzzle.

### 3.4 Estratégia de pesquisa

No desenvolvimento deste trabalho, implementaram-se duas estratégias de etiquetagem distintas.

Para **solução** de tabuleiros estáticos e dinâmicos, não são passadas opções extra no primeiro argumento de **labeling/2**, pois seria expectável apenas uma solução para cada puzzle. Porém, dada a natureza do Skyscrapers e a abordagem de geração utilizada, é possível mais do que uma grelha correta.

Ainda assim, de acordo com a documentação do SICStus Prolog, a opção de retorno da solução mais ótima está ativada por predefinição.

Por outro lado, na **geração** de tabuleiros dinâmicos, esta deverá ser aleatória a cada nova execução do programa. Para tal, concebeu-se o algoritmo abaixo, gerador de uma nova seed, que escolhe aleatoriamente um tabuleiro com o tamanho estipulado.

---

```
1 get_random_label(Var, _, BB, BB1) :-
2     fd_set(Var, Set),
3     select_random_value(Set, Value),
```

---

```

4         (first_bound(BB, BB1), Var #= Value; later_bound(BB, BB1), Var #\= Value).
5
6 select_random_value(Set, RandomValue) :-
7     fdset_to_list(Set, List),
8     length(List, Len),
9     random(0, Len, RandomIndex),
10    nth0(RandomIndex, List, RandomValue).
11 ...
12 labeling([value(get_random_label)], FlatMatrix).

```

---

O valor gerado é posteriormente utilizado nas opções de labeling, resultando num puzzle novo a cada iteração.

## 4 Visualização da solução

É providenciada, ao utilizador, uma interface simples de visualização da solução, cujos predicados encontram-se agrupados no ficheiro **display.pl**.

```

! ?- skyscraper(logicmasters).
Generating Logicmasters puzzle... Done!
Solving... Done!

  5  0  0  2  2  0
0 [ 2  1  3  4  5  6 ] 0
2 [ 3  6  4  2  1  5 ] 0
3 [ 4  5  6  3  2  1 ] 4
4 [ 1  2  5  6  4  3 ] 3
0 [ 5  3  2  1  6  4 ] 2
0 [ 6  4  1  5  3  2 ] 0
  0  3  4  0  0  4

Runtime: 29.522 seconds.

```

**Figura 1.** Representação em Prolog do tabuleiro da Logicmasters.

Como é mostrado acima, o nosso grupo decidiu seguir por uma representação na consola em modo de texto bastante semelhante a como os tabuleiros estão feitos na Logicmasters e no website da BrainBashers. Os números fora da grelha

central correspondem às pistas dadas, que são os edifícios que se conseguem observar quando se olha daquela perspectiva. Quando a pista é 0 significa que não há restrição quanto ao número de edifícios que se conseguem observar. Segue-se abaixo a representação de um outro tabuleiro, com pistas em todas as colunas e linhas.

```
i ?- skyscraper.
Board size to generate: 4.
Generating a 4x4 board... Done!
Generating clues... Done!
Solving... Done!

  2  3  2  1
2 [ 3  1  2  4 ] 1
3 [ 1  3  4  2 ] 2
2 [ 2  4  3  1 ] 3
1 [ 4  2  1  3 ] 2
  1  2  3  2

Runtime: 0.092 seconds.
```

**Figura 2.** Representação em Prolog de um tabuleiro 4x4 gerado aleatoriamente.

O grupo encontra-se satisfeito com o modo como ficou terminada a representação de todos os tabuleiros.

## 5 Resultados

Para analisar o desempenho do algoritmo estabelecido, recorreu-se a múltiplos testes de contagem de tempo para a resolução de grelhas de diferentes tamanhos, **dinamicamente geradas**.

Como expectável, observou-se um crescimento exponencial do tempo de cálculo com o aumento do grau da grelha, sendo que, por exemplo, **tabuleiros 5x5** registaram uma média de **meia décima** de segundo, enquanto que **grelhas 6x6** contabilizavam, aproximadamente, quatro décimas.

Os resultados obtidos apresentam-se na tabela abaixo, tendo a média sido calculada com base em 20 amostras individuais.

**Tabela 1.** Tempo médio de cálculo para diferentes grau de grelha

	<b>3x3</b>	<b>4x4</b>	<b>5x5</b>	<b>6x6</b>
Min	0.023	0.029	0.044	0.047
Max	0.032	0.054	0.079	1.075
Avg	<b>0.026</b>	<b>0.036</b>	<b>0.055</b>	<b>0.376</b>

Suspeita-se que a discrepância dos tempos mínimos e máximos que se realça com o aumento do grau do tabuleiro esteja relacionado com o facto de uma instância de grelha gerada possuir mais (ou menos) soluções.

Após análise cuidada, visto a estratégia de labeling utilizada procurar a primeira solução, constatou-se ser não só expectável, como perfeitamente aceitável esta discrepância observada.

O algoritmo de geração implementado gera pistas em todas as posições, porém, a omissão de algumas destas (enquanto o puzzle se mantenha resolúvel) afeta o tempo de execução na medida em que **diminui a quantidade de soluções possíveis**.

Não existe nenhuma restrição nas regras do puzzle, pelo que preservámos as múltiplas soluções, o que acaba por ser mais interessante para descrever neste relatório.

## 6 Conclusões e trabalho futuro

O grupo conclui que o uso de Prolog com restrições é útil para a resolução de problemas de decisão. Os predicados fornecidos descomplicam algumas tarefas que noutro ambiente nos traria maiores dores de cabeça.

Analisando o projeto, saímos satisfeitos com o resultado final, visto que conseguimos implementar a geração automática de tabuleiros, que funciona sempre, juntamente com a solução do tabuleiro do PDF apresentado.

Com o término deste relatório apresentamos também o término do projeto.

## 7 Bibliografia

- BrainBashers’ Walkthrough:  
<https://www.brainbashers.com/skyscrapershelp.asp>
- Logicmasters India’s Challenge:  
<http://logicmastersindia.com/limitests/dl.asp?attachmentid=659&view=1>.
- SICStus Prolog CLPFD Docs:  
[https://sicstus.sics.se/sicstus/docs/4.1.0/html/sicstus/lib\\_002dclpfd.html](https://sicstus.sics.se/sicstus/docs/4.1.0/html/sicstus/lib_002dclpfd.html)

## 8 Anexo

### 8.1 skyscraper.pl

---

```
1 :- use_module(library(lists)).
2 :- use_module(library(between)).
3 :- use_module(library(clpfd)).
4 :- use_module(library(random)).
5
6 :- include('display.pl').
7 :- include('helpers.pl').
8 :- include('generator.pl').
9
10 :- dynamic clue/1.
11
12 skyscraper :-
13     write('\nBoard size to generate: '),
14     read(BoardSize), % Ask user for a board size to generate.
15
16     format('\nGenerating a ~dx~d board... ', [BoardSize, BoardSize]),
17     generate_full_board(BoardSize, CheatBoard),
18     write('Done!\n'),
19
20     format('Generating clues... ', []),
21     generate_clues(CheatBoard),
22     write('Done!\n'),
23
24     generate_empty_board(BoardSize, Board),
25
26     format('Solving... ', []),
27     solve_board(Board),
```



```

28         append(Board, FlatBoard),
29         start_timer, labeling([], FlatBoard),
30
31         write('Done!\n\n'),
32         display_board(Board, BoardSize), print_timer.
33
34     /**
35      *           Handles static puzzles.
36     **/
37     skyscraper(Puzzle) :-
38         atom_concat(generate_, Puzzle, Generator),
39         call(Generator, Board), length(Board, BoardSize),
40
41         format('Solving... ', []),
42         solve_board(Board),
43         append(Board, FlatBoard),
44         start_timer, labeling([], FlatBoard),
45
46         write('Done!\n\n'),
47         display_board(Board, BoardSize), print_timer.
48
49     /**
50      *           Solves board based on restrictions code.
51     **/
52     solve_board(Board) :-
53         length(Board, Size),
54         declare_board_domain(Board, Size),
55
56         maplist(all_distinct, Board), % Every row has unique values.
57         clue(left-LeftClues), clue(right-RightClues),
58         apply_clues(Board, LeftClues, RightClues),
59
60         transpose(Board, InvertedBoard),
61         maplist(all_distinct, InvertedBoard),
62         clue(top-TopClues), clue(bottom-BottomClues),
63         apply_clues(InvertedBoard, TopClues, BottomClues).
64
65     /**
66      *           Apply clues restrictions.
67     **/
68     apply_clues([], [], []).

```

```
69
70 apply_clues([BH|BT], [CH1|CT1], [CH2|CT2]) :-
71     reverse(BH, BH_Rev),
72
73     CH1 \= 0, get_seen_buildings(BH_Rev, Seen_Rev), Seen_Rev #= CH1,
74     CH2 \= 0, get_seen_buildings(BH, Seen), Seen #= CH2,
75
76     apply_clues(BT, CT1, CT2).
77
78 apply_clues([BH|BT], [_|CT1], [CH2|CT2]) :-
79     CH2 \= 0, get_seen_buildings(BH, Seen), Seen #= CH2,
80     apply_clues(BT, CT1, CT2).
81
82 apply_clues([_|BT], [_|CT1], [_|CT2]) :-
83     apply_clues(BT, CT1, CT2).
```

---

## 8.2 generator.pl

---

```
1 :- use_module(library(lists)).
2 :- use_module(library(clpfd)).
3 :- use_module(library(random)).
4
5 /**
6  *      Generates the tutorial puzzle at https://www.brainbashers.com/skyscrapershelf.asp.
7  *      Uses a 4x4 grid, clues on every direction.
8  */
9 generate_brainbashers(Board) :-
10     write('\nGenerating Brainbashers puzzle... '),
11     retractall(clue(_)), % Clear memory first from previous generations.
12     generate_empty_board(4, Board),
13
14     assertz(clue(left-[1,2,2,2])), assertz(clue(right-[4,3,1,2])),
15     assertz(clue(top-[1,2,3,3])), assertz(clue(bottom-[3,3,1,2])),
16
17     write('Done!\n').
18
19 /**
20  *      Generates the puzzle at http://logicmastersindia.com/lmitests/dl.asp?attachmentid=659&view=
21  *      Uses a 6x6 grid, missing clues on some directions (represented as zeros).
22  */
23 generate_logicmasters(Board) :-
24     write('\nGenerating Logicmasters puzzle... '),
25     retractall(clue(_)), % Clear memory first from previous generations.
26     generate_empty_board(6, Board),
27
28     assertz(clue(left-[0,2,3,4,0,0])), assertz(clue(right-[0,0,4,3,2,0])),
29     assertz(clue(top-[5,0,0,2,2,0])), assertz(clue(bottom-[0,3,4,0,0,4])),
30
31     write('Done!\n').
32
33 /**
34  *      Generates an uninitialized [Size]x[Size] matrix.
35  */
36 generate_empty_board(Size, Matrix) :-
37     bagof(R, Y ^ (between(1, Size, Y), length(R, Size)), Matrix).
38
39 /**
```

```

40  *           Generates a full [Size]x[Size] matrix.
41  *           Based on the randomly generated board, clues are computed and a new puzzle is created.
42  **/
43  generate_full_board(Size, Matrix) :-
44      generate_empty_board(Size, Matrix),
45
46      declare_board_domain(Matrix, Size),
47
48      maplist(all_distinct, Matrix),
49      transpose(Matrix, MatrixRev),
50      maplist(all_distinct, MatrixRev),
51
52      append(Matrix, FlatMatrix),
53      labeling([value(get_random_label)], FlatMatrix). % Generate a random matrix.
54
55  /**
56  *           Generates clues.
57  **/
58  generate_clues(Board) :-
59      retractall(clue(_)),
60      transpose(Board, BoardRev), length(Board, Length),
61
62      generate_left_top_clues(Board, CluesLeft),
63      length(CluesLeft, Length), % Removes the last uninitialized value of list.
64      assertz(clue(left-CluesLeft)),
65
66      generate_right_bottom_clues(Board, CluesRight),
67      length(CluesRight, Length), % Removes the last uninitialized value of list.
68      assertz(clue(right-CluesRight)),
69
70      generate_left_top_clues(BoardRev, CluesTop),
71      length(CluesTop, Length), % Removes the last uninitialized value of list.
72      assertz(clue(top-CluesTop)),
73
74      generate_right_bottom_clues(BoardRev, CluesBottom),
75      length(CluesBottom, Length), % Removes the last uninitialized value of list.
76      assertz(clue(bottom-CluesBottom)).
77
78  /**
79  *           Extracts LEFT/TOP clues from the provided board.
80  **/

```

```

81 generate_left_top_clues([], _).
82
83 generate_left_top_clues([H|T], Clues) :-
84     generate_left_top_clues(T, NewClues),
85     reverse(H, HRev),
86     get_seen_buildings(HRev, Result),
87     append([Result], NewClues, Clues).
88
89 generate_left_top_clues(_, []).
90
91 /**
92  *      Extracts RIGHT/BOTTOM clues from the provided board.
93  */
94 generate_right_bottom_clues([], _).
95
96 generate_right_bottom_clues([H|T], Clues) :-
97     generate_right_bottom_clues(T, NewClues),
98     get_seen_buildings(H, Result),
99     append([Result], NewClues, Clues).
100
101 generate_right_bottom_clues(_, []).
102
103 /**
104  *      Generates a random label for the labeling option on generate.
105  */
106 get_random_label(Var, _, BB, BB1) :-
107     fd_set(Var, Set),
108     select_random_value(Set, Value),
109     (first_bound(BB, BB1), Var #= Value; later_bound(BB, BB1), Var #\= Value).
110
111 select_random_value(Set, RandomValue) :-
112     fdset_to_list(Set, List),
113     length(List, Len),
114     random(0, Len, RandomIndex),
115     nth0(RandomIndex, List, RandomValue).
116
117 /**
118  *      Declares domain on every cell.
119  */
120 declare_board_domain([], _).
121

```

```
122 declare_board_domain([H|T], Size) :-
123     domain(H, 1, Size),
124     declare_board_domain(T, Size).
125
126 /**
127  *      Assigns [Result] to the no. of buildings seen from the left-most position on a list.
128  **/
129 get_seen_buildings([], 0).
130
131 get_seen_buildings([H|T], Result) :-
132     get_seen_buildings(T, TokenResult),
133     maximum(Max, [H|T]),
134     H #= Max #<=> Seen,
135     Result #= TokenResult + Seen.
```

---

### 8.3 display.pl

---

```
1 display_board(Board, BoardSize) :-
2     write(' '), clue(top-TopClues), display_horizontal_clues(BoardSize, TopClues), nl,
3
4     display_top_line(BoardSize),
5     start_iteration(Board, 1, BoardSize).
6
7 start_iteration([], _, BoardSize) :-
8     write(' '), display_bottom_line(BoardSize),
9     write(' '), clue(bottom-BottomClues), display_horizontal_clues(BoardSize, BottomClues), nl.
10
11 start_iteration([H|T], Index, BoardSize) :-
12     clue(left-LeftClues), nth1(Index, LeftClues, LeftClue),
13     format('~d', [LeftClue]),
14     put_code(179), display_line(H, Index, BoardSize), nl,
15
16     NewIndex is Index + 1,
17     start_iteration(T, NewIndex, BoardSize).
18
19 /**
20  *      Displays the horizontal clues.
21  */
22 display_horizontal_clues(_, []).
23
24 display_horizontal_clues(BoardSize, [H|T]) :-
25     format('~d ', [H]),
26     display_horizontal_clues(BoardSize, T).
27
28 /**
29  *      Displays a single line from the board.
30  *      At its last iteration, adds the surrounding board limits.
31  */
32 display_line([], Index, BoardSize) :-
33     length(SpaceList, BoardSize), maplist(=(' '), SpaceList),
34
35     write('\b\b'), put_code(179),
36
37     clue(right-RightClues), nth1(Index, RightClues, RightClue),
38     format('~d', [RightClue]),
39
```

```

40         nl, write(' '),
41         BoardSize \= Index,
42         put_code(179), maplist(write, SpaceList), write('\b\b'), put_code(179).
43
44 display_line([H|T], Index, BoardSize) :-
45     format('~d ', [H]),
46     display_line(T, Index, BoardSize).
47
48 display_line(_, _, _) :- write('\b\b').
49
50 /**
51  *      Displays a stylized bottom and top line for the board.
52  */
53 display_top_line(BoardSize) :-
54     write(' '), put_code(218),
55     length(SpaceList, BoardSize), maplist(=('--'), SpaceList),
56     maplist(write, SpaceList), write('\b\b'), put_code(191), write(' '), nl.
57
58 display_bottom_line(BoardSize) :-
59     put_code(192),
60     length(SpaceList, BoardSize), maplist(=('--'), SpaceList),
61     maplist(write, SpaceList), write('\b\b'), put_code(217), write(' '), nl.
62
63
64 /**
65  *      Prints runtime execution statistics in seconds, rounded to 3 decimal places.
66  */
67 start_timer :- statistics(walltime, _).
68 print_timer :-
69     statistics(walltime, [_ , T]),
70     format('\nRuntime: ~3f seconds.\n\n', [T / 1000]).

```

---



## 8.4 helpers.pl

---

```
1  replace([_|T], 0, New, [New|T]).
2
3  replace([H|T], Index, New, [H|R]) :-
4      I1 is Index - 1,
5      replace(T, I1, New, R).
6
7  clear_console :-
8      clear_console(40), !.
9
10 clear_console(0).
11
12 clear_console(N) :-
13     nl, N1 is N - 1, clear_console(N1).
```

---