Skyscraper

Andr Cruz[1] and Edgar Carneiro[2]

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal,
feup@fe.up.pt,
WWW home page: http://www.fe.up.pt

Abstract. This article was written for the course unit "Logic Programming", from the course Master in Informatics and Computing Engineering. This article purpose is to present how the program we developed is able to solve the decision problem that is the puzzle Skyscraper, independently of board size. The program developed is also able to generate Skyscraper puzzles as well. The program developed uses Logic Programming with Constraints as the approach to solve and generate the puzzles.

Keywords: skyscraper, CLP, sicstus, PLOG, FEUP

1 Introduction

The main purpose of this project was to develop a program that would be able to solve either Decision Problems or Optimization Problems, by using Logic Programming with Constraints.

Our group was assigned with a decision problem: the Skyscraper puzzle. Succinctly, the puzzle consists of a grid that the player needs to fill, according to some given restrictions, and without repeating the same digits per column and per row.

The article approaches several topics such as: what were the variables used and its domains, what were the constraints used and their implementation in the program, what is the labeling strategy implemented, what are the results of the developed program and what are the final conclusions obtained from developing the project.

2 Problem Description

Skyscraper consists of a puzzle where the player needs to fill a grid with digits from 1 to N — with N being the size of the grid — where each row and column contains each digit exactly once. In the grid, each number represents the height of a building. The numbers outside the grid indicate how many buildings can be seen when looking from that direction. Taller buildings block the view of smaller buildings meaning that every number beyond a number bigger than it will not be taken into account, when looking from that direction.

The difficulty of the puzzle arises from the conjugation of two factors: the digits in rows and columns must all be distinct and the restrictions outside the grid.

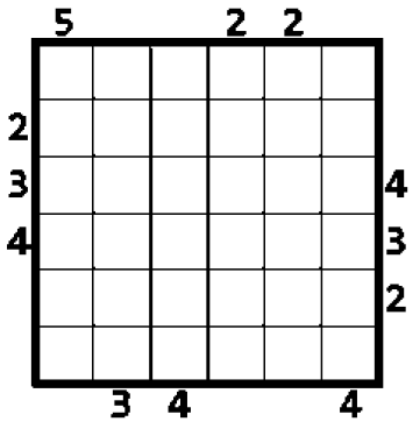

Fig. 1. Unsolved 6x6 Skyscraper puzzle

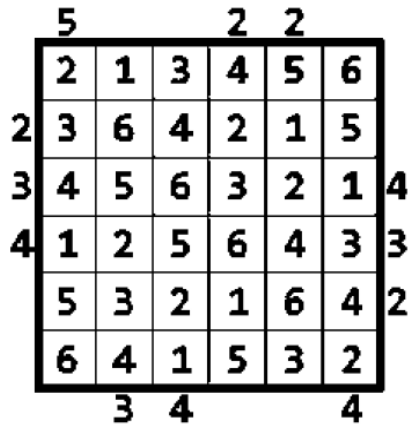

Fig. 2. Solved 6x6 Skyscraper puzzle

3 Approach

In the implementation of Skyscraper in Prolog, the group used a list of lists to represent the grid. The value of each element will correspond to the height of the building it represents. Elements whose value is not known will be represented by a '_', therefore representing in Prolog non- instantiated values.

For the restrictions outside the grid, the group also decided to use a list of lists. The list containing the other lists will always have length four since each one of its elements is a list containing the restrictions of the correspondent border. The order by which the border's restrictions are kept in the list is: Top border restrictions, Left border restrictions, Bottom border restrictions and Right border restrictions. If the restriction for a certain row or column does not exist, a '0' is used in its representation.

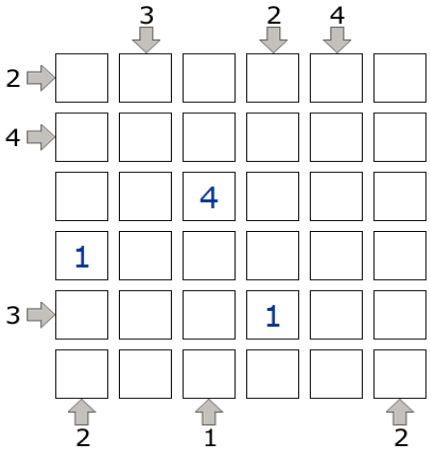

Fig. 3. Example of a skyscraper puzzle followed by its Prolog representation

Listing 1.1. Prolog grid representation

```
testBoard([
  [_, _, _, _, _, _],
  [_, _, _, _, _, _],
  [_, _, 4, _, _, _],
  [1, _, _, _, _, _],
  [_, _, _, 1, _, _],
  [_, _, _, _, _, _]
]).
```

Listing 1.2. Prolog representation of border restrictions

```
testRestrictions([
  [0, 3, 0, 2, 4, 0],
  [2, 4, 0, 0, 3, 0],
  [2, 0, 1, 0, 0, 2],
  [0, 0, 0, 0, 0, 0]
]).
```

3.1 Decision Variables

The decision variables associated with a skyscraper puzzle are: the lists representing the rows of the grid. Since all the elements in a row and column must have a different value — as it is a skyscraper rule — the domain of the elements in each row will have to be defined between 1 and the length of the board, N, and they will also have to be all distinct. The following code presents the application of the referred restrains to the decision variables.

Listing 1.3. Row domain restrictions

```
restrictBoardDomain([], _).
restrictBoardDomain([Row |
    Board], N) :-
  length(Row, N),
  domain(Row, 1, N),
  all_distinct(Row),
  restrictBoardDomain(Board, N).
```

Listing 1.4. Column all elements distinct

```
all_distinct_columns(_, 0) :- !.
all_distinct_columns(Board, N) :-
  N > 0, !,
  getBoardCol(Board, N, Col),
  all_distinct(Col),
  NewN is N - 1,
  all_distinct_columns(Board, NewN).
```

TODO Os side domains tb sao? penso que nao porque nao queremos instanciar nada ai... although it seems in code

3.2 Constraints

The restrictions defined for the puzzle are the ones from the rules. The rules that force the inexistence of repeated elements in either a row or a column were already implemented by the way the domains are defined (see section **Decision Variables**). Therefore, the challenge involving the project was the addition of PROLOG restrictions to implement the missing rule: assuring that the number outside the grid would indeed control the number of 'visible buildings'.**TODO - Ver esta frase, if we did** In the end, some restrictions were also added in order to increase the efficiency of the implemented solution.

Theoretically, what we needed to implement in order to achieve the last rule was an IF CLAUSE, being that: if the element is bigger than the maximum value so far, than the element is the new maximum value, and the elements correspondent building is visible, otherwise, the building is not visible and the maximum value stays the same. However, since in Constraints Logic Programming the elements value would not be instantiated this implementation proved hard. **Um bcd bullshit este texto, mudar? xD ou mm apagar**

The solution we came up to implement that last rule was a *logic OR*, because the element being analyzed was either a maximum value and the number of visible buildings would increment, or it was not and the count of visible buildings would stay the same. In the end, we wanted to, when the line was finished being analyzed, the number of visible buildings would be equal to the corresponding border restraining value. The predicate would be called for each line/ column

for each of the four possible directions: left to right, right to left, top to bottom and bottom to top.

Listing 1.5. Constraint that assures correct number of visible buildings

```
/**
 * Apply restrictions to Row.
 * +Predicate Order in which elements are analyzed - fetches an element.
 * +Num is the number of visible skyscrapers according to the above
     order.
 */
applyToRow(Num, Row, Max, GetElement) :-
  call(GetElement, Row, El, RemainderRow),
  NewNum #>= 0,
  (El #> Max #/\ NewMax #= El #/\ NewNum #= Num - 1) #\/
  (El #=< Max #/\ NewMax #= Max #/\ NewNum #= Num),
  applyToRow(NewNum, RemainderRow, NewMax, GetElement).
applyToRow(0, [], _, _).
```

TODO? 'restries rgidas e flexveis' - n encontrei nada sobre isto em pdfs ou lado algum, perguntei e tb nao arranjei quem soubesse. caguei e meti so restries por geral

3.3 Evaluation Function

In Skyscraper there is no evaluation of solutions, since the puzzle is solved whenever a solution is found.

3.4 Search Strategy

The labeling strategy implemented in the program was the ***ffc***, also known as *most_constrained*. This labeling strategy makes use of the most constrained heuristic: ' a variable with the smallest domain is selected, breaking ties by (a) selecting the variable that has the most constraints suspended on it and (b) selecting the leftmost one'[3]. We opted for this heuristic as it would be the one providing faster and more efficient solutions for the skyscraper puzzle. This was natural having in mind the kind ofconstraints that were applied during the development, namely the constraint that assure the building heights were correct,**ver a frase a partir daqui :/** that is: since the entire board is not instantiated the solution will be faster if the first most constrained values — therefore the ones most unlikely to wrong — are first discovered and snowball from there on.

4 Solution Presentation

For the solution presentation in text mode the predicate *printBoard* is used. If run as ***printBoard(+Board)***, it only prints the given board in a user friendly

way. However, if run as ***printBoard(+Board, +Restrictions)*** it will print the board in a user friendly way while also displaying the boarder restrictions being applied to each row or column. The predicate makes use of helper predicates such as ***printRow(+Row)***, that prints the given row and ***printHBorder(+Length)***, that prints the horizontal border at the top and the bottom. All this predicates are defined in the file ***display.pl***.

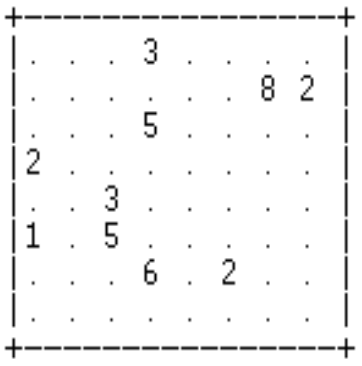

Fig. 4. Example of a call to printBoard/1

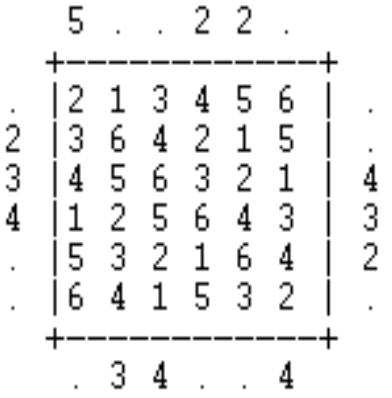

Fig. 5. Example of a call to printBoard/2 with a solved board

5 Results

The group tried to make extensive and exhaustive tests to be able to conclude fiercely about the given results.

The results of several tests that the group made the program went through were:

Table 1. Each of the values is the result of the average of several tests made

Board Size	Time /s	Backtracks
4x4	0.030	9
5x5	0.087	659
6x6	0.751	5438
7x7	12.489	159196
8x8	48.239	315893

In the end, the conclusions the group came up with were:

- There is an exponential relation between the board size and time taken to solve the puzzle. For smaller boards the difference in board size affects slightly the difference of times (board size 4 to baoard size 5, differences of around 0.5s). However in bigger boards the difference of board sizes affect time fiercely (board size 7 to board size 8, differences of arount 30s). **meter aqui o grafico catita**
- There is a strong correlation between the number of backtracks (and the other statistics, such as *Resumptions, Entailments, Prunings and Constraints created*) and time. The higher the alue of the statistics the higher the tame ita took the program to solve the puzzle. **meter aqui utroo grafico catita**
- Also worth notice, despiting not being visible in the table because its values are an average, is the fact that puzzle with a higher number of border restricitons are faster to solve. This is naturally explained since the information received by the solver is bigger and therefore the number of backtracks done is smaller.

6 Conclusion and Future Work

We believe that our knowledge about Logic Programming was deeply increased, with everything we had proposed to do being accomplished in the required time.

The program developed has no limitations regarding board size or problem generation, despite becoming slower with bigger board sizes. Therefore, the only way possible to improve the developed application would be by improving the solver efficiency and time. However, the group tried fiercely to make that improve

by trying out several approaches but in the end, the best solution was the one being presented.

The group also agreed in the fact that Logic Programming with Constrains revealed itself to be an extremely powerful tool, able to solve easily certain problems (such as the one this work is based in) that would take much more time and effort to solve with other paradigms.

To sum up, despite Logic Programming with Constraints being a totally different paradigm from what the group had ever worked with, we quickly got used to it and learnt to appreciate the cons and pros that it presents, thus having it culminate in a project we are proud of.

References

1. L. Sterling, E. Y. Shapiro, and D. H. D. Warren, The Art of Prolog: Advanced Programming Techniques. MIT Press, 2010.
2. M. Carlsson and T. Fruhwirth, SICStus Prolog User's Manual. SICS Swedish ICT AB, 4.3.5 ed., 2016.
3. Skyscraper rules, http://logicmastersindia.com/lmitests/dl.asp?attachmentid=659&view=1