LightBulbs - Constraint Logic Programming

Ivo Saavedra - up201707093 and João Cardoso - up201806531
FEUP-PLOG, 3MIEIC01, Grupo LightBulb

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

Abstract. This article contains the implementation details of the application developed for the second assignment for the Logical Programming subject. The goal of this project was to develop a program capable of creating and solving every instance of the Light bulb puzzle. The goal of this puzzle is to find every lit light bulb, considering that a light bulb is only lit if and only if the number inside it is equal to the number of lit neighboring lamps (including itself).

Keywords: Prolog · SICStus · Light Bulbs.

1 Introduction

This article was developed as complement for the second practical assignment of the Logical Programming subject of the 3rd year of the MIEIC course. The goal of this project was to develop an application capable of creating and solving "Light bulb" type puzzles using the SISCstus prolog development system along with the restriction tools provided by the CLPFD library. The objective of these puzzles is to determine which light bulbs inside a n*n square are turned on. A light bulb is only on when the number of adjacent bulbs (including itself) equals the number attributed to it. This document is organized in the following manner:

- **Problem Description:** detailed description of the problem being analyzed
- **Approach:** section describing the implementation of the application
  - **Decision Variables:** description of the decision variables and their domains
  - **Constraints:** details of the rigid and flexible constraints
- **Solution Presentation:** description of the adopted solution presentation
- **Experiments and Results:**
  - **Dimension Analysis:** results obtained after testing boards of different sizes
  - **Search Strategies:** result comparison of different heuristics search functions
- **Conclusions**
- **References**

2 Problem description

The Light bulb puzzle consists of a two dimensional board consisting of n[1*] light bulbs per line and m[1*] light bulbs per column, where each light bulb has a number on it. Each light bulb is on if and only if it's number is equal to the number of lit neighboring (directly or diagonally adjacent) light bulbs, including itself. When given a board we must determine all it's possible solutions; the number of solutions may range from one to many, however there are cases in which the board is impossible to solve. Having that in consideration we classified the Light Bulbs puzzle as a decision problem.

3 Approach

3.1 Decision Variables

For this puzzle the decision variables are the values inside each cell of the input board. As previously stated, each board cell as a value representing the number of adjacent bulbs that need to be lit in order for the current cell to be as well. Taking into consideration that every cell can at most have eight adjacent cells and that for the bulb to be lit we need to have N adjacent lit bulbs plus the current one we come to the conclusion that the maximum number inside a light bulb is 8+1=9. As for the minimum value we came to the conclusion that it should be 1. If the value inside a bulb was equal to zero, then that bulb would always be turned off, because of the constraints of this puzzle.

3.2 Constraints

Elements
Our problem is a constraint satisfaction problem, therefore, every constraint is a hard constraint, which always have to be true (if they are called). We have a "results list", where each element is a variable that can be 1, if the corresponding light bulb is turned off or, otherwise, 0. This list is a flattened version of the list that is returned at the end. It's first element is at line 1 column 1, the second is at line 2 column 1, etc... [Note: we use nth1, so the first element will be index 1.]

Element searching
As previously mentioned, a bulb is only lit when the number inside it is equal to the number of adjacent light bulbs (including itself) that are lit. In order to accomplish this we iterated over all of the input board's cells and for each cell we fetched every adjacent variable. The maximum number of adjacent cells is eight, so for each light bulb we needed eight variables. If the current cell had less than eight neighbors, for instance, if it was in a corner, then the adjacent variables that were out of bounds would take the value of zero. After having gotten all the adjacent variables we started applying the constraints. The first

one being that the sum of all the adjacent variables should be different from the number inside the current bulb. For example, being given a 4x3 board: We are using a flattened list, therefore, the line length is 4 and the existing elements are 1 through 16. The program starts by checking for the first (1 - top left) cell, it first checks for the cell above and to the left of it first (1 - 4 - 1 = -4). Since that element isn't in the board, the would be element's value is zero. The same thing happens for the element above, above (1 - 4 = -3), above to the right (1 - 4 + 1 = -2) and center to the left (1 - 1 = 0), and, later on, to the element under and to the left. There isn't a check for itself, since, if an element exists, then it exists. The other elements are assigned to the corresponding variables (2nd, 5th and 6th elements of the "results list").

Since we repeat the steps above for each list element, when it gets to the end, every element will check weather each adjacent light bulb (including itself) are lit up.

Sum condition

If the sum of every element in the "results list" is 0, then the "results list" is automatically invalid.

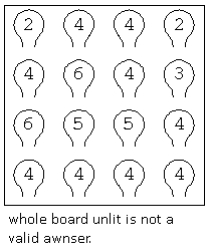

Fig. 1. No whole board unlit example

Lockout condition
First we check if the sum of the neighboring elements, excluding itself, is equal to the number on the light bulb. If it is, then the current "results list" is automatically invalid.

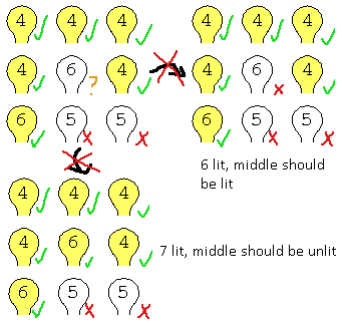

Fig. 2. Lockout condition example

Light condition
For each element, we get the sum of the the neighboring elements, excluding itself, plus one (the +1 is because we are assuming that the light bulb itself is on for this case). If that sum is equal to the number on the light bulb we're checking, then the light bulb can be on, but doesn't necessarily have to be.

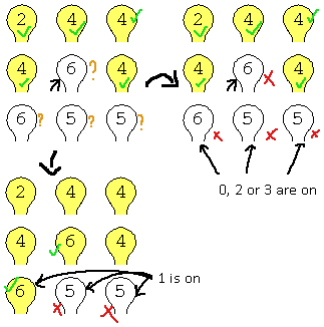

Fig. 3. Light condition example

Each ligh bulb can always be off, unless it meets the "sum condition" or the "lockout condition".

These restrictions are used both to obtain the solution and to create the problem using a solution.

4 Solution Presentation

In order to make the solution more intelligible, the input board is displayed along with the result board, in which the latter only contains the cells that have a lit bulb, allowing for an easier understanding of the solution. The elapsed time is also displayed for the experiments and results section. The predicates responsible for displaying the results are called after the solution is obtained. As each board can originate more than one result the solution consists of a list of matrices, each one representing the possible output boards for the original board. The **showResults** predicate is responsible for iterating over the result matrices and displaying each one in the console.

```
/* SHOW RESULTS -------------------------------------------------*/
showResults([], _, _).
showResults([First|Rest], Original, N) :-
   separator(N),
   showResult(Original, First, N),
   showResults(Rest, Original, N).

showResult([], [], _).
showResult([OriginalHead | OriginalRest], [OnHead | OnRest], N) :-
   nl, write('| '), showResultLine(OriginalHead, OnHead),
   separator(N),
   showResult(OriginalRest, OnRest, N).

showResultLine([], []).
showResultLine([FirstOriginal | RestOriginal], [FirstRes | RestRes]) :-
   FirstRes =:= 1,
   format('~p | ', [FirstOriginal]),
   showResultLine(RestOriginal, RestRes).

showResultLine( [_ | RestOriginal], [_|RestRes]) :-
   write('  | '),
   showResultLine(RestOriginal, RestRes).
```

Fig. 4. display result predicates

```
ORIGINAL ------
| 2 | 4 | 4 | 3 |
| 4 | 3 | 6 | 4 |
| 4 | 8 | 6 | 6 |
| 2 | 3 | 4 | 3 |

+---------------+
|   |   | 4 | 3 |
+---------------+
|   | 3 |   | 4 |
+---------------+
|   |   | 6 |   |
+---------------+
|   | 3 | 4 | 3 |
+---------------+

 [!] Elapsed time: 9
---
```

Fig. 5. solution display example

5 Experiments and Results

5.1 Dimension Analysis

The Dimension Analysis graph indicates how our program scales with board size. Board size indicates the length and width of the square board. Time indicates the time it takes to solve it. We used a randomly created board with at least one solution. As expected, the time to solve increased with the board size, however, the spike from 7 to 8 was unexpected. Despite that, the time increase relative to the board size increase seemed to stabilize after that, so we'll assume the spike was an anomaly. We did one test per board size.

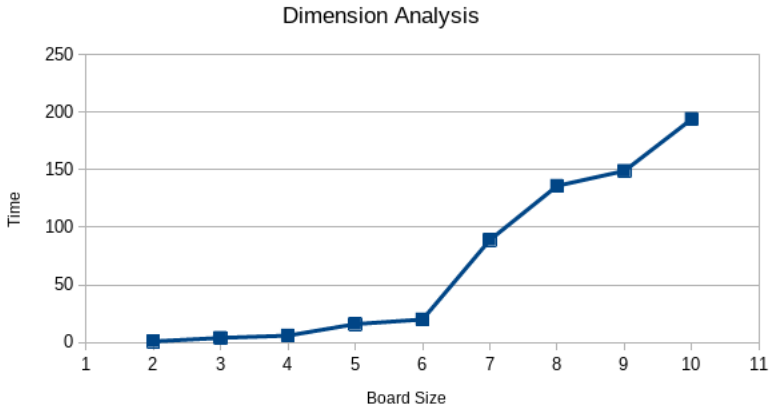

Fig. 6. Effect of the board size on execution time

5.2 Search Strategies

We ran every our application with different search options but obtained similar results to the ones displayed on figure. 6. These options didn't have that much of an effect on our tests because most off them affect the domain of the variables which in our case was between 0 and 1. Another reason for this, was the fact that this puzzle represents a "satisfaction" problem in which we, for every board, calculate all the possible solutions, therefore the order in which they originated didn't affect the execution time.

6 Conclusions

The objective of this project was to develop an application capable of creating and solving light bulb puzzles. At the end of this assignment we feel that our knowledge on the CLPFD prolog module has definitely increased as well as our experience with constraint logical programming.
We observed, as was expected, that the execution time of the application increased with the size of the board being solved. The results showed that in spite of the rapid increase of the execution time between the board sizes of 7 and 8 the graph was mostly linear which indicates that our implementation of this problem was acceptable.
As far as the required features we implemented others like reading boards from a file, generating random possible light bulb puzzles and checking weather an input board is valid. The main flaw of our application would be the overuse of if statements which are prone to failures and maybe aren't that efficient.
Taking all these factors into consideration we believe that this project was a success as we were able to develop an application capable of generating and solving light bulb type puzzles.

References

1. SICstus Documentation, https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html. Last accessed 4 Jan 2020
2. SWI Prolog Documentation, https://www.swi-prolog.org/. Last accessed 4 Jan 2020
3. Light Bulb puzzles, https://erich-friedman.github.io/puzzle/bulb/. Last accessed 4 Jan 2020