# Restriction problem resolution*

Marcia Sofia de Sousa Meira[1][0000−0001−8038−3256] and Diogo Alexandre Silva Teixeira[2][0000−0002−1573−7553]

[1] Faculdade de Enhenharia, Rua Roberto Frias, 4200-465, Porto, Portugal
[2] Prolog
[3] Turma 1
[4] Grupo Lol Sudoku 3

**Abstract.** In this article, we will talk about the resolution of the problem presented to us in this subject, as well as the ways to improve it.

**Keywords:** Prolog · Restrictions · Puzzles.

## 1 Introduction

The objectives of this work, as proposed by the teachers, lay in the resolution of a restriction problem, in this case a board game, using the Prolog language. The chosen game was Lol-Sudoku. This article will be structured in the following fashion, as specified by the teachers:

- Introduction
- Problem description
- Approach
  - Decision Variables
  - Constraints
  - Evaluation Function
  - Search Strategy
- Solution Presentation
- Results
- Conclusions and Future Work
- References
- Annex

## 2 Problem Description

The chosen problem was Lol-Sudoku, a board game similar to classic Sudoku. This puzzle consists in the filling of columns, rows and areas with a set of numbers, all of them distinct, and the attribution of areas in the end, since only one of the areas is defined in the beginning of the puzzle. The size of the puzzle can vary between 4x4 and 8x8.

* Supported by FEUP

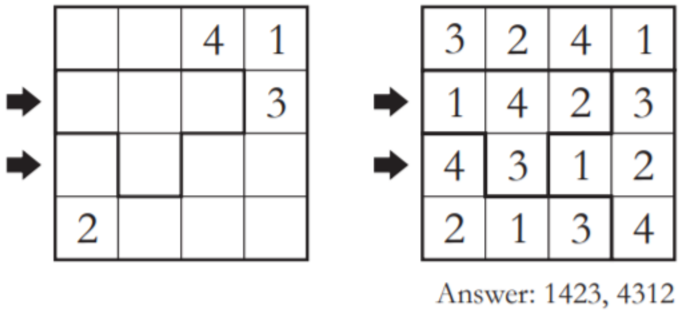


**Fig. 1.** Example of a solution

## 3 Approach

In the approach to this problem, we tried to make the most of the restrictions we had access with the Prolog language, using a PSR modulation. A PSR is modulated using the variables, their domains and the used restrictions.

### 3.1 Decision Variables

In order to solve this problem, we had to define a set of decision variables. First of all, each element of the resulting matrix has a domain between 1 and the size of the puzzle, and they are then arranged into lists, representing the rows of the puzzle. In order to define the areas in the end, we made another matrix, also arranged into list representing rows, and variable domain between 1 and the size of the puzzle.

It is important to note that the final number of areas and the number of their elements is always the same as the size of the given puzzle.

### 3.2 Constraints

After the definition of the variables, we proceed to define the constraints.

- All elements in a row of the Sudoku puzzle must be distinct
- All elements in a column of the Sudoku puzzle must be distinct
- All elements in an area of the Sudoku puzzle must be distinct
- Each element in the area map is only presented a number of times equal to the size of the puzzle

### 3.3 Evaluation Function

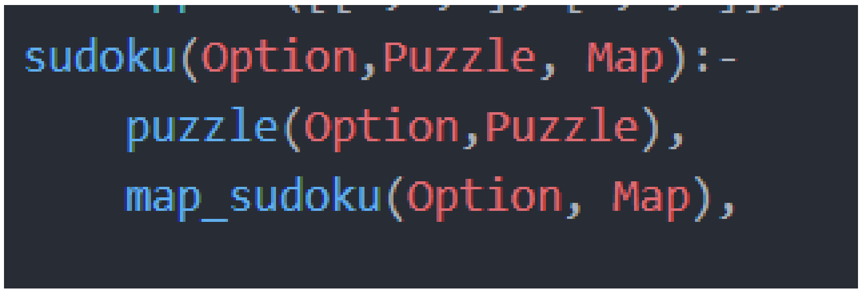
As an argument in the beginning of the program, the user must choose one of the example puzzles. If the user chooses to make a puzzle of its own, he must add it directly to the code. We also expect the user to give us two lists of atoms, both with the number of atoms equal to the size of the puzzle, that will be filled with the solution of the puzzle and the area map, respectively. The program itself revolves around the given matrix, presenting solutions not only for the Sudoku puzzle, but for the final areas as well. When there are no solutions for the given matrix, the program replies with ”no”. If the puzzle has a valid solution, both the Sudoku puzzle matrix and the area map are presented in the screen, with the possibility of checking other solutions by entering ”;” in the console terminal. Unfortunately, we could not find a way to stop the program from presenting an area map that only differs from the previous one in the area names, and not in their configuration.

### 3.4 Search Strategy

After the definition of both the decision variables and constraints, we used the labelling predicate to find possible solutions to the puzzles. We present the predicate with a flatten list of both the Sudoku puzzle matrix and the area map matrix.

## 4 Solution Presentation

The solution is returned by the main predicate, *sudoku(+Option,-Puzzle, -Map)*, as illustrated below.



```
sudoku(Option,Puzzle, Map):-
    puzzle(Option,Puzzle),
    map_sudoku(Option, Map),
```

**Fig. 2.** Main Predicate

## 5 Results

In order to test the code, we added a set of example puzzles and area maps, with varying sizes, such as 4x4, 6x6 and 8x8. We also used the predicate statistics(+Key, -Value) in order to calculate the execution time of the program. After testing and collecting the execution times, we were able to elaborate the following graphic.

There is only an example of a complex puzzle because, taking in consideration the lack of puzzles of this kind present in the web, and the difficulty in creating bigger solvable puzzles, we choose to use as example puzzles with more regular shapes. Even so, after we studied the graph and the information acquired, we presume more complex puzzles will still follow a similar behaviour, making us believe the program has a time and space complexity of *O(n)*, making it a linear complexity problem.

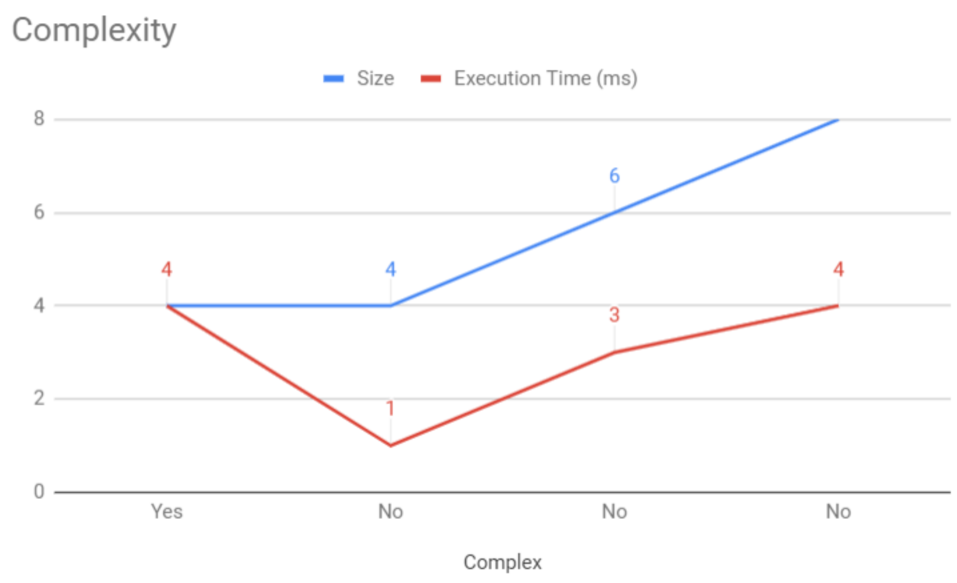


**Fig. 3.** Complexity Graph

## 6 Conclusions and Future Work

Working on this project, we could experience first hand the use of restrictions and how effective they are at solving this kind of puzzles, although some of the problems we faced while making the program had an inherent logic that was definitely harder to elaborate, in order to arrive at the right restriction. In the future, we would also like to be able to solve a problem that still lingers in the program: the multiple solutions presented in the end, regarding the area map, that are all symmetric and end up being irrelevant. Besides that, we think the program is well elaborated and all flaws that it presents are minor ones, none of them slowing its execution nor making it lack solutions.

## 7 References

## References

1. Project PDF, http://logicmastersindia.com/lmitests/dl.asp?attachmentid=738. Last accessed 23 Dec 2018

## 8 Annex

### 8.1 lol.pl

```
:- use_module(library(lists)).
:- use_module(library(clpfd)).
:- include('auxiliary.pl').
/*
    Puzzle templates
*/
puzzle(p1,Puzzle):-
    Puzzle = [
        [_,_,4,1],
        [_,_,_,3],
        [_,_,_,_],
        [2,_,_,_]
    ].
puzzle(p2, Puzzle):-
    Puzzle = [
        [_,2,_,_],
        [_,1,4,_],
        [_,_,3,_],
        [_,_,_,_]
    ].
puzzle(p3, Puzzle):-
    Puzzle = [
        [_,_,_,_,_,_,5,_],
        [_,8,_,1,7,_,_,4],
        [6,_,_,4,_,_,_,_],
        [_,_,_,_,4,_,6,3],
        [_,_,_,_,_,8,_,7],
        [_,_,4,_,6,_,_,_],
        [_,_,5,_,_,6,_,2],
        [2,_,_,_,_,_,3,_]
    ].
puzzle(p4, Puzzle):-
    Puzzle = [
        [_,_,_,1,_,6],
        [6,_,4,_,_,_],
        [1,_,2,_,_,_],
        [_,_,_,5,_,1],
        [_,_,_,6,_,3],
        [5,_,6,_,_,_]
    ].

/*
    Map Templates
*/
map_sudoku(p1, Map) :-
    Map = [
        [_,_,_,_],
        [1,1,1,_],
        [_,1,_,_],
        [_,_,_,_]
    ].
map_sudoku(p2, Map):-
    Map = [
        [1,1,_,_],
        [1,1,_,_],
        [_,_,_,_],
        [_,_,_,_]
    ].
map_sudoku(p3, Map):-
    Map = [
        [1,1,1,1,_,_,_,_],
        [1,1,1,1,_,_,_,_],
        [_,_,_,_,_,_,_,_],
        [_,_,_,_,_,_,_,_],
        [_,_,_,_,2,2,2,2],
        [_,_,_,_,2,2,2,2],
        [_,_,_,_,_,_,_,_],
        [_,_,_,_,_,_,_,_]
    ].
map_sudoku(p4, Map):-
    Map = [
        [_,_,_,_,_,_],
        [1,1,1,_,_,_],
        [1,1,1,_,_,_],
        [_,_,_,_,_,_],
        [_,_,_,_,_,_],
        [2,2,2,2,2,2]
    ].

test(AreaValues):-
```

```
    append([[1,2,3], [4,5,6]],AreaValues).
sudoku(Option,Puzzle, Map):-
    puzzle(Option,Puzzle),
    map_sudoku(Option, Map),

    /*
    Start Timer
    */
    statistics(walltime, [_TimeSinceStart |
     ↪  [_TimeSinceLastCall]]),

    length(Puzzle, Length),
    Lsquared is Length*Length,
    length(AreaValues, Lsquared),

    append(Map, FlatMap),
    append(Puzzle, FlatPuzzle),

    domain(FlatPuzzle, 1, Length),

    create_global_list(Length, GlobalList),
    global_cardinality(FlatMap, GlobalList),

    adjacent(Map, Length),
    transpose(Puzzle, Columns),

    sync_puzzle_map(Length, FlatPuzzle, FlatMap, AreaValues),
    all_distinct(AreaValues),
    maplist(all_distinct, Puzzle),
    maplist(all_distinct, Columns),

    append(FlatPuzzle, FlatMap, PandM),

    labeling([],PandM),
    statistics(walltime, [_NewTimeSinceStart | [ExecutionTime]]),
    write('Execution took '), write(ExecutionTime), write(' ms.'),
     ↪  nl.

adjacent(Map, Length) :-
    adj_aux_x(1, Map, Length).

adj_aux_x(Length, Map, Length):-
    adj_aux_y(Length, 1, Map, Length).
adj_aux_x(X, Map, Length):-
    X #> 0,
    X #< Length,
    adj_aux_y(X, 1, Map, Length),
    XX #= X + 1,
    adj_aux_x(XX, Map, Length).

adj_aux_y(X, Length, Map, Length):-
    get_member(X, Length, Map, Value),
    check_adjacency(X, Length, Map, Value, Length).
adj_aux_y(X, Y, Map, Length) :-
    Y #> 0,
    Y #< Length,
    get_member(X, Y, Map, Value),
    check_adjacency(X, Y, Map, Value, Length),
    YY #= Y + 1,
    adj_aux_y(X, YY, Map, Length).

check_adjacency(X, Y, Map, Value, _Length) :-
    XX #= X - 1,
    XX #> 0,
    get_member(XX, Y, Map, Value).
check_adjacency(X, Y, Map, Value, Length) :-
    XX #= X + 1,
    XX #=< Length,
    get_member(XX, Y, Map, Value).
check_adjacency(X, Y, Map, Value, _Length) :-
    YY #= Y - 1,
    YY #> 0,
    get_member(X, YY, Map, Value).
check_adjacency(X, Y, Map, Value, Length) :-
    YY #= Y + 1,
    YY #=< Length,
    get_member(X, YY, Map, Value).

sync_puzzle_map(_, [], [], []).
sync_puzzle_map(N, [PH|PT], [MH|MT], [AH|AT]):-
    N*(MH-1) + PH mod N #= AH,
    sync_puzzle_map(N, PT, MT, AT).
```

## 8.2 Auxiliary.pl

```
:- use_module(library(clpfd)).

/* my_nth -> nth1 but works with elements being other lists */
my_nth(1, [X|_T], X).
my_nth(N, [_H|T], X):-
    N #> 1,
    NN #= N - 1,
    my_nth(NN, T, X).

/*get_member in matrix*/
get_member(X, Y, List, Value):-
        get_row(Y, List, L),
        get_column(X, L, Value).
get_row(1,[L|_], L).
get_row(Y, [_|Tail], L):-
        YY #= Y -1,
        get_row(YY, Tail, L).
get_column(1, [Value|_], Value).
get_column(X, [_|Tail], Value):-
        XX #= X - 1,
        get_column(XX, Tail, Value).

/* create_global_list_aux -> for global_cardinality - if puzzle
 ↪  size is N, list is of form [1..N - N]*/
create_global_list(Length, List):-
    create_global_list_aux(1, Length, List).
create_global_list_aux(Length, Length, [Length-Length|[]]).
create_global_list_aux(N, Length, [N-Length|T]):-
    NN #= N+1,
    create_global_list_aux(NN, Length, T).

/* label_all -> calls label to all lists in given list */
label_all([]).
label_all([H|T]):-
    labeling([],H),
    label_all(T).
```