Skyscraper
Decision Problem solved with Constraint Logic Programming

Andre Cruz[1] and Edgar Carneiro[2]

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal,
`FEUP-PLOG, Class 3MIEIC01, Group Skyscraper_5`

Abstract. This article presents a Constraint Logic Programming approach to define and ultimately solve the puzzle "Skyscraper". The purposed solution is independent of board size and number of restrictions. The solver can also be presented with a partially-filled board, as is customary in larger boards. We also dynamically generate "Skyscraper" puzzles, of different sizes and with varying difficulty. Results show that "Skyscraper" is a prime example for being solved with CLP, as the solution's code is brief, efficient, and simple to understand.

Keywords: Skyscraper, Constraint Logic Programming, Sicstus, PLOG, FEUP

1 Introduction

The "Skyscraper" puzzles consists of an NxN grid with some clues along its sides. This type of puzzles combines the row and column constraints of Sudoku with external clue values that re-imagine each row or column of numbers as a road full of skyscrapers of varying height.

We model the presented puzzle as a Constraint Satisfaction Problem (CSP), therefore representing it by a set of domain variables, their domains, and a set of restrictions.

Constraint Logic Programming (CLP) is a merger of two declarative paradigms: constraint solving and logic programming. Viewing the subject rather broadly, constraint logic programming can be said to involve the incorporation of constraints and constraint solving methods in a logic-based language. We refer the reader to [1] for more on this subject.

In this paper, section 2 begins by describing the problem at hand in detail. Section 3 presents our approach to solving this decision problem using Prolog, as well as the finer grained details of the CSP modelling. The choice of labeling strategy is also explained. Section 4 describes how we generate test instances. Section 5 describes the presentation of the solution in a user-friendly manner. Section 6 presents our results performance-wise, as well as a set of key issues we discovered while developing the code. Finally, section 7 presents the conclusion and future work.

2 Problem Description

The "Skyscraper" puzzle consists of an NxN grid with some clues along its sides. The objective is to place a skyscraper (represented by its height) in each square, with a height between 1 and N, so that each row and column contains each digit exactly once. In addition, the number of visible skyscrapers, as viewed from the direction of each clue, is equal to the value of the clue. Note that higher skyscrapers block the view of lower skyscrapers located behind them. A comprehensive booklet with the puzzle's rules can be found here [2].

The difficulty of the puzzle can vary according to three factors: the size of the board, the number of restrictions along the side of the board, and the possible hints in the board itself (it is common to present larger boards partially filled).

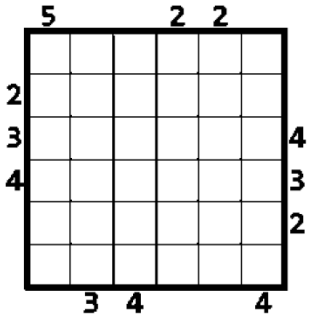

Fig. 1. Unsolved 6x6 Skyscraper puzzle

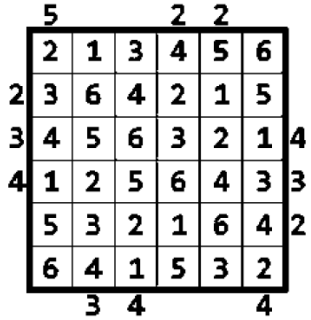

Fig. 2. Solved 6x6 Skyscraper puzzle

3 Approach

We present a Constraint Logic Programming approach to define and ultimately solve the puzzle "Skyscraper". In order to accomplish this, we use the clp(fd) — Constraint Logic Programming over Finite Domains — library in sicstus Prolog. For more information on this prolog implementation, and said library, we refer the reader to [3] and [4].

Regarding the implementation of Skyscraper in Prolog, we decided to represent the board's matrix as a list of lists, whose elementary units are integer numbers, representing the height of the skyscraper in that position. Elements whose value is not known are represented by a '_', therefore representing in Prolog non-instantiated values.

For the restrictions along the side of the grid, we use a list of lists, the first of which being the top restrictions, and the rest following in counter-clockwise order. If the restriction for a certain row or column is undefined, a '0' is used in its position.

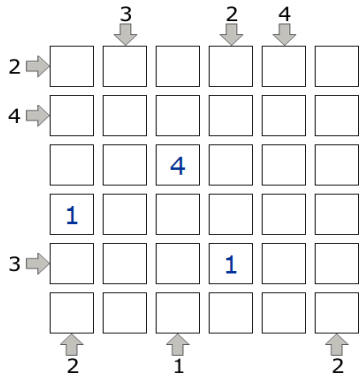

Fig. 3. Example of a skyscraper puzzle followed by its representation in PROLOG

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 numbers of the board's grid, and the numbers of the restrictions along the side of the grid. The domain of the elements of the grid, according to the puzzle's rules [2], is between 1 and N. The domain of the restrictions along the side is also between 1 and N. Because this restrictions can be defined, we extended this domain to include the value 0, as this value internally represents an undefined restriction.

The application of domain restrictions is done in the following code.

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).
```

Listing 1.5. Constraints for the Border restrictions domains

```
sidesDomain(Size, [List | Rest]) :-
  length(List, Size),
  domain(List, 0, Size),
  sidesDomain(Size, Rest).
sidesDomain(_, []).
```

3.2 Constraints

The restrictions defined for the puzzle in Prolog are fairly faithful to the ones from the rules. The uniqueness of variables along their columns and rows is fairly straightforward to implement using the *all_distinct* predicate (see the listings 1.3 and 1.4 in 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'.

The solution we came up to implement that last rule was a *logic XOR*, #\: the element being analyzed was either the maximum so far, and the following buildings with smaller height wouldn't count towards the side restriction; or it was not a maximum building, therefore not visible along the side, and the count of visible buildings would stay the same.

When the row/column being analyzed had no more elements, the number of visible buildings (a domain variable in itself) would be equal, #=, to the corresponding side 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 1.6. 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, [], _, _).
```

3.3 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’[4]. We opted for this heuristic as it would be the one providing faster and more efficient solutions for the skyscraper puzzle.

Naturally the default labeling options were not appropriate for this representation, as the order in which the variables were presented had no intrinsic value (labeling left-to-right was not particularly useful). Having in mind the kind of constraints used in the modeling of the problem, a certain value was more likely to stick (to not backtrack) if its variable had a tighter domain; and, if it did backtrack, it would do so less often.

4 Dynamic Generation of Puzzle Instances

In order to appropriately test our solver, we decided to implement predicates to dynamically generate puzzle instances.

As the solver itself contains a logical definition of the problem at hand, generating examples was a simple case of finding solutions without instantiating the restrictions along the side of the board.

The problem that followed was that a board with all the restrictions undefined technically fulfills all restrictions. In order to solve this we adopted the following approach: for each side restriction, there’s a probability it will be defined, and if it is its domain cannot feature the value 0 (which represents an

undefined restriction). In this manner we can easily generate easy boards (with some restrictions), or hard boards (with a lot of restrictions).

The following code is high-level view of the implemented functionality, the full code can be viewed in **generator.pl** (available in the appendix).

```
generateBoard(Size, Board, Sides, Probability) :-
  setrand(100),
  length(Sides, 4),
  sidesDomain(Size, Sides),
  restrictVarsInSides(Sides, Probability),

  solveBoard(Size, Board, Sides).
```

5 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***.

```
+-----------------+
|. . . 3 . . . . |
|. . . . . . 8 2 |
|. . . 5 . . . . |
|2 . . . . . . . |
|. . 3 . . . . . |
|1 . 5 . . . . . |
|. . . 6 . 2 . . |
|. . . . . . . . |
+-----------------+
```

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

```
    5 . . 2 2 .
  +-------------+
. |2 1 3 4 5 6 | .
2 |3 6 4 2 1 5 | .
3 |4 5 6 3 2 1 | 4
4 |1 2 5 6 4 3 | 3
. |5 3 2 1 6 4 | 2
. |6 4 1 5 3 2 | .
  +-------------+
    . 3 4 . . 4
```

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

6 Results

The group led extensive and exhaustive tests to be able to extract comprehensive conclusions regarding the obtained results.

The overall results, after averaging several test run-times, are presented in table 1. Note that the 8x8 results consist of solving partially filled boards, as is customary in boards this size. The blank boards take in average 5 times longer to solve.

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

Board Size	**Time /s**	**Backtracks**
4x4	0.0	9
5x5	0.047	659
6x6	0.291	5438
7x7	10.489	159196
8x8	18.239	315893

In the end, the group concluded that:

– There is an exponential relation between the board size and time taken to solve the puzzle.

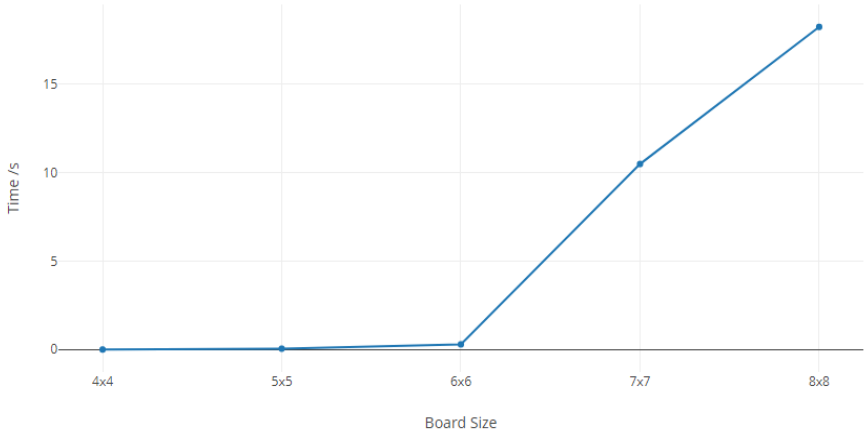

Fig. 6. Relation between board size and time taken to find a solution.

– There is a strong correlation between the number of backtracks (and *Resumptions, Entailments, Prunings*) and run-time. As was expected, a higher number of backtracks leads to a higher run-time for the solver.

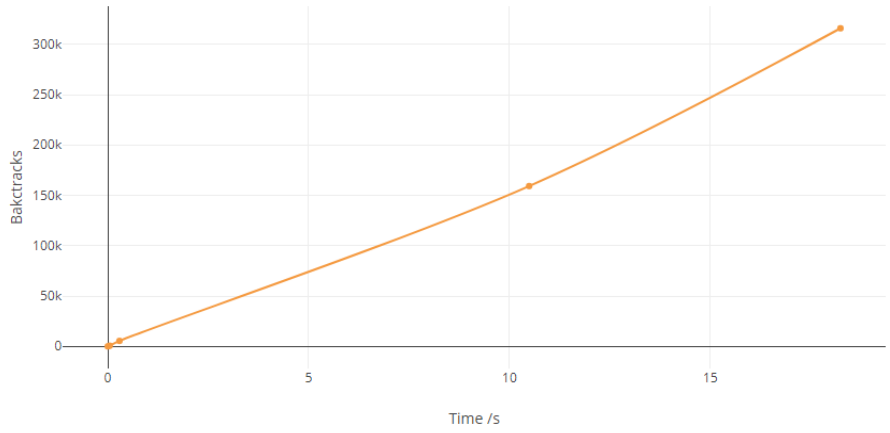

Fig. 7. Relation between number of backtracks and time to solve the puzzle

– It is also worth noticing that, for the same board size, a puzzle with a higher number of side restrictions is faster to solve. This was expected, as the board's domain variables have stricter constraints, leading to a decreased number of backtracks.

7 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 evidently slower with larger board sizes. Therefore, the only way possible to improve the developed application would be by improving the solver's efficiency. However, the group tried fiercely to make that improve by trying out several approaches, but in the end the best solution was the one presented here.

It is also evident that Constraint Logic Programming revealed itself to be an extremely powerful tool, extremely appropriate to tackle problems like the one presented, resulting in a faithful and straightforward representation of the problem, and much faster development.

To sum up, despite Constraint Logic Programming being a unique programming paradigm, rather distinct from classical ones, 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 to present.

8 Acknowledgements

This article was written for the course unit “Logic Programming”, from the Master in Informatics and Computing Engineering, a course in the Faculty of Engineering of the University of Porto.

References

1. J. Jaffar and M. J. Maher, "Constraint logic programming: a survey," *The Journal of Logic Programming*, vol. 19-20, p. 503581, 1994.
2. "Skyscrapers - puzzle contest."
3. L. Sterling, E. Y. Shapiro, and D. H. D. Warren, *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 2010.
4. M. Carlsson and T. Frhwirth, *SICStus Prolog Users Manual*. SICS Swedish ICT AB, 4.3.5 ed., 2016.

A Appendix

skyscraper.pl

```
:- include('solver.pl').
:- include('display.pl').
:- include('test_data.pl').
:- include('generator.pl').

%Predicate to run the default puzzle
skyscraper :-
  nl, write('Skyscraper!'), nl, nl,
  testRestrictions(R),
  R = [Up | _Rest],
  length(Up, Size),
  solveBoard(Size, B, R),
  printBoard(B, R).
```

solver.pl

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

/**
 * Domain restriction
 */
sidesDomain(Size, [List | Rest]) :-
  length(List, Size),
  domain(List, 0, Size),
  sidesDomain(Size, Rest).
sidesDomain(_, []).

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

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).


% Get the nth1 column of the given Board (1 indexed)
getBoardCol([], _, []).
getBoardCol([Row | Board], N, [El | Col]) :-
  element(N, Row, El),
  getBoardCol(Board, N, Col).


%% Gets the FIRST element of a given row
getFirstElement([Element | RemainderRow], Element, RemainderRow).

%% Gets the LAST element of a given row
getLastElement(Row, Element, RemainderRow) :-
  append(RemainderRow, [Element], Row).


%% LEFT to RIGHT
applyLeftToRight(Num, Row) :-
  applyToRow(Num, Row, 0, getFirstElement).
```

```
%% RIGHT to LEFT
applyRightToLeft(Num, Row) :-
  applyToRow(Num, Row, 0, getLastElement).

/**
 * 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, [], _, _).

/**
 * Applies restrictions Horizontally along the board
 * +Predicate is the predicate used to apply restrictions on the fetched
     Row.
 */
applyAllHorizontalRestrictions([0 | Rest], [_ | Rows], Predicate) :-
  applyAllHorizontalRestrictions(Rest, Rows, Predicate).
applyAllHorizontalRestrictions([N | Rest], [Row1 | Rows], Predicate) :-
  call(Predicate, N, Row1),
  applyAllHorizontalRestrictions(Rest, Rows, Predicate).
applyAllHorizontalRestrictions([], [], _).

/**
 * Applies restrictions Vertically along the board
 * +Predicate is the predicate used to apply restrictions on the fetched
     Column.
 */
applyAllVerticalRestrictions(Restrictions, Board, Predicate) :-
  applyAllVerticalRestrictions(Restrictions, Board, Predicate, 1).
applyAllVerticalRestrictions([0 | Rest], Board, Predicate, Count) :-
  NewCount is Count + 1,
  applyAllVerticalRestrictions(Rest, Board, Predicate, NewCount).
applyAllVerticalRestrictions([N | Rest], Board, Predicate, Count) :-
  NewCount is Count + 1,
  getBoardCol(Board, Count, Col),
  call(Predicate, N, Col),
  applyAllVerticalRestrictions(Rest, Board, Predicate, NewCount).
applyAllVerticalRestrictions([], _, _, _).


/**
```

```
 *  +Sides -> a list of lists, each of which represents the restrictions
     on the side of the board (number of visible buildings).
 *      -> in order: [TopRestrictions, LeftRestrictions,
     BottomRestrictions, RightRestrictions]
 *      -> elements of list are in range [0,N], 0 meaning an undefined
     restriction
 *      -> elements correspond to restrictions in top->bottom (left/right
     lists) or left->right (top/bottom lists) order
 *  -Board -> a list of lists (a matrix)
 */
solveBoard(Size, Board, Sides) :-
  Sides = [Top, Left, Bottom, Right],
  sidesDomain(Size, Sides),

  % Domain
  length(Board, Size),
  restrictBoardDomain(Board, Size),
  all_distinct_columns(Board, Size),

  % Apply restrictions to board rows/columns
  applyAllHorizontalRestrictions(Left, Board, applyLeftToRight),
  applyAllHorizontalRestrictions(Right, Board, applyRightToLeft),
  applyAllVerticalRestrictions(Top, Board, applyLeftToRight),
  applyAllVerticalRestrictions(Bottom, Board, applyRightToLeft),

  append(Board, FlatBoard),
  append(Sides, FlatSides),
  append(FlatBoard, FlatSides, DomainVariables),
  labeling([ffc], DomainVariables).
```

generator.pl

```
:- use_module(library(lists)).
:- use_module(library(random)).
:- include('solver.pl').

% Restrict the number of constraints shown , according to the probability
restrictVarsInSides([], _).
restrictVarsInSides([First | Rest], Probability) :-
  restrictVarsInRow(First, Probability),
  restrictVarsInSides(Rest, Probability).

% Restrict the number of constraints shown , according to the probability
restrictVarsInRow([], _).
restrictVarsInRow([V1 | Vars], Probability) :-
  maybe(Probability), !,
  V1 #\= 0,
  restrictVarsInRow(Vars, Probability).
restrictVarsInRow([_ | Vars], Probability) :-
  restrictVarsInRow(Vars, Probability).

/**
 * Generates boards of differente difficulty according the given
     probability.
 * +The probability will control the amount of Restrictions that are
     shown.
 */
generateBoardEasy(Size, Board, Sides) :-
  generateBoard(Size, Board, Sides, 0.95).
generateBoardMedium(Size, Board, Sides) :-
  generateBoard(Size, Board, Sides, 0.8).
generateBoardHard(Size, Board, Sides) :-
  generateBoard(Size, Board, Sides, 0.65).
generateBoard(Size, Board, Sides, Probability) :-
  setrand(100),
  length(Sides, 4),
  sidesDomain(Size, Sides),
  restrictVarsInSides(Sides, Probability),

  solveBoard(Size, Board, Sides).

% Size = 8, generateBoardEasy(Size, B, S), printBoard(B, S),
    solveBoard(Size, SB, S), printBoard(SB, S).
% Size = 6, generateRandomBoard(Size, Board, Sides), printBoard(Board,
    Sides), solveBoard(Size, SameBoard, Sides), Board = SameBoard.
```

display.pl

```
%Dictionary for user friendly visualization of elements
translate(0, '.').
translate(P, P).

% printBoard(+Board)
%% prints the given board on the screen
printBoard(Board) :-
  Board = [Row | _],
  length(Row, RowLength),
  printHBorder(RowLength),
  printBoardAux(Board),
  printHBorder(RowLength).

printBoardAux([]) :- !.
printBoardAux([Row | Board]) :-
  printRow(Row), nl,
  printBoardAux(Board).

% printBoard(+Board, +Restrictions)
%% prints the given board, and all the provided side restrictions
printBoard(Board, Restrictions) :- !,
  Restrictions = [Top, Left, Bottom, Right],
  write('  '), printRowAux(Top), nl,
  length(Top, RowLength),
  write(' '), printHBorder(RowLength),
  printBoard(Board, Left, Right),
  write(' '), printHBorder(RowLength),
  write('  '), printRowAux(Bottom), nl, nl.

% printBoard(+Board, +LeftRestrictions, +RightRestrictions)
%% prints the given board, and the provided side restrictions for left
    and right
printBoard([], [], []) :- !.
printBoard([Row | Board], [L1 | Left], [R1 | Right]) :-
  translate(L1, SymbL), translate(R1, SymbR),
  write(SymbL), write(' '),
  printRow(Row),
  write(' '), write(SymbR), nl,
  printBoard(Board, Left, Right).

% printRow(+Row)
%% prints the provided list/row and adds '|' after and before the list
printRow(Row) :-
  write('|'),
  printRowAux(Row),
  write('|'), !.
printRowAux([]) :- !.
```

```
printRowAux([El | Row]) :-
  translate(El, Symb),
  write(Symb), write(' '),
  printRowAux(Row).

% printHorizontalBorder(+Length)
%% prints the top or bottom border for the board, example of a boarder:
    '+-------+'
printHBorder(Length):-
  write('+'),
  printHBorderAux(Length),
  write('+'), nl, !.
printHBorderAux(0) :- !.
printHBorderAux(Length) :-
  write('--'),
  NewLength is Length - 1,
  printHBorderAux(NewLength).
```

test_data.pl

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

/** Test Boards **/
%% https://www.brainbashers.com/skyscrapers.asp

%% 6 by 6 board -- given example -- 166199 backtracks, 6311 with ffc
testRestrictions([
  [5, 0, 0, 2, 2, 0],
  [0, 2, 3, 4, 0, 0],
  [0, 3, 4, 0, 0, 4],
  [0, 0, 4, 3, 2, 0]
]).

%% 5 by 5 board
testRestrictions2([
  [4, 0, 1, 2, 3],
  [0, 2, 0, 4, 0],
  [0, 0, 4, 0, 0],
  [0, 0, 0, 0, 2]
]).

%% 4 by 4 board
testRestrictions3([
  [4, 0, 0, 2],
  [0, 3, 0, 0],
  [0, 0, 4, 0],
  [0, 0, 0, 3]
]).

%% 8 by 8 board
testRestrictions4([
  [0, 0, 5, 3, 0, 2, 0, 4],
  [3, 3, 0, 3, 0, 3, 0, 0],
  [2, 4, 0, 0, 4, 0, 0, 0],
  [0, 0, 2, 0, 4, 4, 0, 1]
]).

board4([
  [_, _, _, 3, _, _, _, _],
  [_, _, _, _, _, _, 8, 2],
  [_, _, _, 5, _, _, _, _],
  [2, _, _, _, _, _, _, _],
  [_, _, 3, _, _, _, _, _],
  [1, _, 5, _, _, _, _, _],
  [_, _, _, 6, _, 2, _, _],
  [_, _, _, _, _, _, _, _]
]).
```

```
%% 6 by 6 board
testRestrictions5([
  [0, 1, 3, 0, 0, 0],
  [0, 0, 4, 2, 2, 0],
  [3, 0, 0, 0, 0, 0],
  [3, 3, 2, 0, 2, 4]
]).

board5([
  [1, _, _, _, _, _],
  [_, 4, _, _, _, _],
  [_, _, _, _, _, _],
  [_, _, 2, _, _, _],
  [_, _, _, _, _, _],
  [_, _, _, _, _, _]
]).

%% 7 by 7 board
testRestrictions6([
  [0, 0, 0, 0, 0, 3, 4],
  [0, 0, 4, 2, 0, 0, 5],
  [0, 2, 0, 2, 0, 4, 0],
  [0, 0, 0, 2, 5, 2, 0]
]).

board6([
  [_, _, _, _, _, _, _],
  [4, 3, _, _, _, _, _],
  [2, _, _, _, _, _, 1],
  [_, _, _, _, _, _, _],
  [_, _, _, _, _, _, _],
  [_, _, 1, _, _, _, _],
  [_, _, _, 3, _, _, _]
]).

% 8 by 8 board
testRestrictions7([
  [0, 0, 5, 3, 0, 2, 0, 4],
  [3, 3, 0, 3, 0, 3, 0, 0],
  [2, 4, 0, 0, 4, 0, 0, 0],
  [0, 0, 2, 0, 4, 4, 0, 1]
]).

board7([
  [_, _, _, 3, _, _, _, _],
  [_, _, _, _, _, _, 8, 2],
  [_, _, _, 5, _, _, _, _],
  [2, _, _, _, _, _, _, _],
  [_, _, 3, _, _, _, _, _],
```

```
  [1, _, 5, _, _, _, _, _],
  [_, _, _, 6, _, 2, _, _],
  [_, _, _, _, _, _, _, _]
]).

% Functions to get duration time
reset_timer:- statistics(walltime, _).
print_time:-
  statistics(walltime, [_, T]),
  TS is ((T/10)*10)/1000,
  nl, write('Time: '), write(TS), write('s'), nl.

% Print stats for the given restricitons and the given, if HasBoard is
    'yes'
% Otherwise only Restrictions are used and board starts empty
% getTestStats(+Restrictions, +HasBoard, +Board)
getTestStats(Restrictions, no, _):-
  call(Restrictions, R),
  R = [Up | _Rest],
  length(Up, Size),
  reset_timer,
  solveBoard(Size, _B, R),
  write('Solved for Restrictions: '), write(Restrictions), nl,
  print_time, fd_statistics, nl.

getTestStats(Restrictions, yes, Board):-
  call(Restrictions, R),
  R = [Up | _Rest],
  length(Up, Size),
  call(Board, B),
  reset_timer,
  solveBoard(Size, B, R),
  write('Solved for Restriction: '), write(Restrictions), nl,
  write('Solved for Board: '), write(Board), nl,
  print_time, fd_statistics, nl.

%Prints the stats of each test
getAllTestsStats:-
  getTestStats(testRestrictions, no, _),
  getTestStats(testRestrictions2, no, _),
  getTestStats(testRestrictions3, no, _),
  getTestStats(testRestrictions4, yes, board4),
  getTestStats(testRestrictions5, yes, board5),
  getTestStats(testRestrictions6, yes, board6),
  getTestStats(testRestrictions7, yes, board7).
```