Square Puzzle Problem Solution with Constraint Programming

Ana Inês Barros (up201806593) and Eduardo Correia (up201806433)

FEUP-PLOG
MIEIC04 - Square_1
Faculty of Engineering of University of Porto

Abstract. This project was developed during the course of *Logic Programming*, using *SICStus Prolog* as its development environment.
The goal of the project was to solve a decision and optimization problem using constraint logic programming.
The chosen problem in question was to solve *Square* puzzle game problems.

Keywords: Decision problems · Logic programming · Constraint programming · Prolog.

1 Introduction

This project was developed in the 3rd year *Logic Programming* course unit of the *Integrated Master in Informatics and Computing Engineering* of the *Faculty of Engineering of University of Porto*.

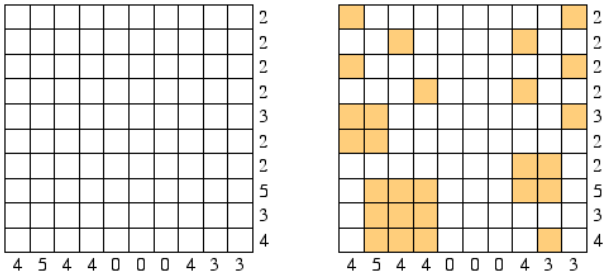
The main goal of this project was to build a program, using *Constraint Logic Programming*, to solve the *Square* puzzle game, developed by Mathematics Professor, Erich Friedman.

This article has the following structure:

- **Problem Description:** Detailed puzzle description and rules.
- **Approach:** Problem description modulation as a constraint satisfaction problem
- **Solution Presentation:** Explanation of predicates used to present the solution and output examples.
- **Experiments and Results:** Dimensional analysis and search strategies.
- **Conclusion and Future Work:** Our expectations and thoughts.
- **References:** Books, articles and web pages consulted during the development of this project.

2 Problem Description

A *Square* puzzle consists of a grid with the same number of rows and columns. Initially, all grid squares are blank, the only exception is if they are marked with



(a) Unsolved puzzle (b) Solved puzzle

Fig. 1: Example of a puzzle

an X. Each row and column is associated with a number, that ranges from 0 up to the square's side length.

This number indicates how many grid cells should be filled in the correspondent row or column.

The grid must only be filled with squares (of any size) that do not touch or overlap, even at the corners (hence the name) *Square*.

3 Approach

We had two distinct approaches for this problem, both of which didn't fully succeeded. However, we will describe the thinking process behind each and their respective flaws.

In the first approach, the goal was to place the squares using the predicate `disjoint2`.

However, this solution demands to know the number of squares *a priori*, since `disjoint2` can only work with a predetermined list of squares. Or else, we need to restrict the number of squares between 0 and number of cells divided by four (since they are all disjoint) but that solution is not viable for larger inputs.

The second approach, used a matrix of 1s (filled cells) and 0s (blank cells), however we couldn't fully restrict the cells to be composed of squares, since this would be to implement a similar version of `disjoint2` by ourselves.

3.1 Decision Variables

The puzzle input hints are equal for both approaches and it consists of two lists, each one containing the number of filled cells each row/column must have in each index, by order.

First Approach Three lists were used to represent the placement of the squares. Two lists are used to represent the row and the column where a square starts. The other list is used to represent the size of each square.

Second Approach A matrix, i.e. a list of lists, with size N by N, was used to represent the board state, with each inner list's domain being between 0 (blank cell) or 1 (filled cell). Here is the code.

```
generate_grid(Grid, Size) :-
    generate_grid(Grid, Size, Size).

generate_grid([], _, 0).

generate_grid([GridRow|T], Size, Counter) :-
    C is Counter - 1,
    length(GridRow, Size),
    domain(GridRow, 0, 1),
    generate_grid(T, Size, C).
```

3.2 Constraints

First Approach In this approach, we constraint the square placement in such a manner that, the squares are all disjoint (that is they never touch).

```
construct_squares(Size, StartsX, StartsY, SquareSizes, Squares),

disjoint2(Squares, [margin(0, 0, 1, 1)]),
```

And also, the sum of the sizes of the squares that intersects a given row or column corresponds to the number of filled cells of that same row/column.

```
lines_constraints(_, [], _, _).

lines_constraints(Index, [NumFilledCells|T], Starts, SquareSizes) :-
    line_constraints(Index, NumFilledCells, Starts, SquareSizes),
    I is Index + 1,
    lines_constraints(I, T, Starts, SquareSizes).

line_constraints(Index, NumFilledCells, Starts, SquareSizes) :-
    (
        foreach(Start, Starts),
        foreach(SquareSize, SquareSizes),
        foreach(Usage, Usages),
        param(Index)
    do
        Intersect #<=> (
            Start #=< Index #/\
            Index #< Start + SquareSize
        ),
        Usage #= Intersect * SquareSize
    ),
    sum(Usages, #=, NumFilledCells).
```

Second Approach There were mainly three sets of constraints we applied in this approach to solve the problem.

The first two were pretty straightforward. To make sure each row and column had the exact necessary number of filled squares we simply enforced it to be equal to the sum of the correspondent line cells, since the filled cells are represented with ones in the matrix.

```
line_constraints([], []).

line_constraints([FilledCells|T1], [GridLine|T2]) :-
    sum(GridLine, #=, FilledCells),
    line_constraints(T1, T2).
```

$$\sum_{i=1}^{SquareSize} Cell(i) = RowNumber \tag{1}$$

$$\sum_{j=1}^{SquareSize} Cell(j) = ColumnNumber \tag{2}$$

The last constraint, however, was more tricky and we couldn't achieve to transpose our logic into Prolog. However, the constraints were defined as the following.

For each cell of the board, if it is an upper left corner then it should be a square.

```
square_constraint(Size, _, _, _, Size).                % Reached end of grid

square_constraint(I, Size, Rows, Columns, Size) :-     % Reached end of row
    NewI is I + 1,                                     % Skip to next row
    square_constraint(NewI, 0, Rows, Columns, Size).

square_constraint(I, J, Rows, Columns, Size) :-
    is_upper_left_corner(I, J, Rows, IsUpperLeftCorner),
    is_square(I, J, Rows, Columns, Size, IsSquare),
    IsUpperLeftCorner #=> IsSquare,

    NextJ is J + 1,                                    % Next Cell
    square_constraint(I, NextJ, Rows, Columns, Size).  % Recursion
```

For a cell to be an upper left corner, the cell above it, the cell by its left side and the cell above it and on the left side of it must all be blank.

```
is_upper_left_corner(I, J, Rows, IsUpperLeftCorner) :-
    get_cell(I, J, Rows, Cell),  % Get current cell
```

```
    TopI is I - 1,
    get_cell(TopI, J, Rows, TopCell),            % Get Top cell

    LeftJ is J - 1,
    get_cell(I, LeftJ, Rows, LeftCell),          % Get Left cell

    get_cell(TopI, LeftJ, Rows, TopLeftCell),    % Get upper left cell

    (
        Cell #= 1 #/\
        TopCell #= 0 #/\
        LeftCell #= 0 #/\
        TopLeftCell #= 0
    ) #<=> IsUpperLeftCorner.
```

To constraint a set of cells to be a square we used a set of materialisable constraints. The border around the left and top of the squares must be all composed of blank cells and the interior of the square is composed of filled cells.

```
is_square(I, J, Rows, Columns, Size, IsSquare) :-
    square_line(I, J, Rows, Size, Width, 1),

    TopI is I - 1,
    LeftJ is J - 1,

    square_line(TopI, J, Rows, Size, BorderWidth, 0),
    square_line(LeftJ, I, Columns, Size, BorderHeight, 0),

    BottomI is I + 1,

    Before #<=> Width #>= 1,

    square_interior(BottomI, J, Rows, Size, Width, Before, 0),

    IsSquare #<=> (
        Before #/\
        BorderWidth #>= Width #/\
        BorderHeight #>= Width
    ).
```

The core predicate for this is the `square_line/8`, that restricts a group of contiguous cells in a line starting at coordinates I, J to all have the specified value.

This was both useful to restrict the top and left borders of the square, since they must be a group of cells with the same size of the square but filled with 0s and also to restrict the interior of the square.

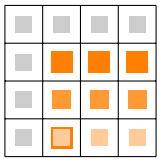


Fig. 2: Caption

```
% Gets square size with top left corner at I row and J column

square_line(I, J, Rows, Size, Length, Value) :-
    square_line(I, J, Rows, Size, 1, 0, Length, Value).

% Reached end of line - update length

square_line(_, Size, _, Size, _, Length, Length, _).

% Count filled consecutive cells

square_line(I, J, Rows, Size, CellBefore, Counter, Length, Value) :-
    get_cell(I, J, Rows, Cell),
    IsValue #<=> ((Cell #= Value) #/\ CellBefore),

    NewCounter #= Counter + IsValue,
    NewJ is J + 1,

    square_line(I, NewJ, Rows, Size, IsValue, NewCounter, Length, Value).
```

The `square_interior/7` is very similar to the `square_line/8` predicate, working as a 2D version of it.

In `square_line/8`, only a line is restricted. In `square_interior/7`, a whole array of lines is restricted, so that there is a contiguous set of contiguous lines, all with the same length and filled with 1s, effectively having a square shape.

```
square_interior(Size, _, _, Size, 0, _, 0).

square_interior(Size, _, _, Size, Width, _, Counter) :-
    Counter #= Width - 1.

square_interior(I, J, Rows, Size, Width, Before, Counter) :-
    square_line(I, J, Rows, Size, Length, 1),

    IsSquareLine #= ((Length #= Width) #/\ Before),
    NewCounter #= Counter + IsSquareLine,
    NewI is I + 1,
```

```
square_interior(NewI, J, Rows, Size, Width, IsSquareLine, NewCounter).
```

4 Solution Presentation

Since we used Unicode characters to represent the cells, we recommend using a mono spaced font, such as *Deja Vu Sans Mono*. We print information of the time it took to solve, the list of squares and the result grid.
This is an example of the solution presentation:

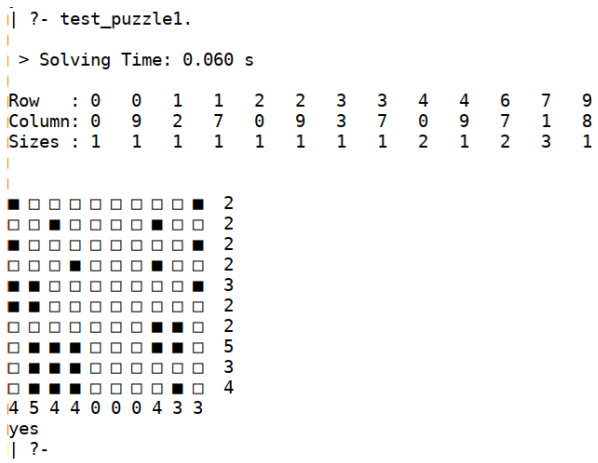


Fig. 3: Solution output

5 Experiments and Results

5.1 Dimensional Analysis

To test our program, we used the set of tests on the website `https://erich-friedman.github.io/puzzle/squa`
We registered each of the inputs into the file puzzles.pl. In the file tests.pl, we can run each test and check if the answer.

To run a test:

```
?- consult(.../Square/tests.pl).
?- test_puzzle1.
```

For each test, our program is able to get a solution but with different times. The solution is always unique.

Example:

```
?- test_puzzle1
Solving Time: 0.02 s

?- test_puzzle2
Solving Time: 1.300 s
```

5.2 Search Strategy

We tried some different search strategies and these were the results:

Option	Time(s)
LeftMost	0.02
ff	0.02
ffc	0.05
up	0.03
down	1.51
step	0.02
enum	0.03
bisect	0.03

We did not test with neither max nor min because only one solution is given. This results were obtained by running test_puzzle1.

6 Conclusions and Future Work

In summary, in this project, by using SICStus Prolog CLPFD library we learned a new paradigm to solve problems, Constraint Logic Programming, which is widely used in optimization and scheduling problems, not only in Prolog but in other languages as well.

It was really hard to shift from the paradigms we're usually used to this one. Because of that and due to the problem's complexity, we couldn't fully implement a correct solution.

However, because of that same reason, we had to research and really push ourselves with our logic and reasoning to try different approaches and in the process we learned a lot about the different areas of constraint programming while trying to implement a solution and formulate the problem.

References

1. sigarra.up.pt (2020). FEUP - Logic Programming. [online] Available at: https://sigarra.up.pt/feup/en/ucurr_geral.ficha_uc_view?pv_ocorrencia_id=459482 [Accessed 28 Dec. 2020]
2. Friedman, E. (2011). Square Puzzles. [online] Available at: https://erich-friedman.github.io/puzzle/square/ [Accessed 28 Dec. 2020]

3. stackoverflow.com (2020). Square Puzzle Problem Solution with Constraint Programming. [online] Available at: https://stackoverflow.com/questions/65497652/square-puzzle-problem-solution-with-constraint-programming [Acessed 02 Jan. 2020]