Easy As ABC using CLP

Márcio Fontes, Rui Andrade

FEUP-PLOG, 3MIEIC3, Group Easy_As_ABC_3

Abstract. Boarding games are extremely useful to apply methods of problems optimization and solving them as efficiently as possible. The present article reflects the approach for the solving of a boarding game called *Easy As ABC* using CLP[1]. It was proved that there is no optimal solution for this problem, i.e, finding a solution *A* is not faster than finding a solution *B*. Also, it was clear that with the developed project, it was possible to find a solution for the max. board size in just 0.2 secs.

1 Introduction

The solving of a boarding game is within *The Problem Solving Paradox*, since it's needed to be both methodical and creative enough to solve problems. The first part of problem solving, defining and exploring the problem requires a methodical left-brain approach. On the second part it's required to come up with ideas and solutions, which means being creative, i.e, using our right-brain. Since a computer can't (for now) do that, it's motivating to develop a project using Logic Programming to find a solution as fast as possible.

The main objetive of the developed project is to find a solution of a given empty board (with pre-defined constraints or not) and this article describes the approach, solution presentation and the observed results, along with the arrived conclusions and future work.

2 Problem Description

Easy As ABC is a boarding game of size NxN, whose objetive is to find a possible solution/board based on constraints indicated by the player. The board cells are initially all blank. If the player doesn't specify any kind of constraints, then there are many solutions. However, if the player specifies any constraints there may be more than one solution, only one solution, or even no solution. The game rules are specified below along with an illustrative figure.

Put a letter in the range indicated into some of the cells. Each row/column must have each letter exactly once. A letter outside the board tells the first letter seen from that direction, looking into the board.

[1] CLP - Constraint Logic Programming

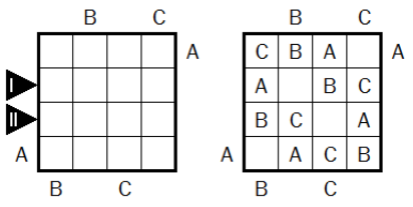

Figure 1. Initial Board with Constraints and Final Solution

Let N be the board's size, then there are $N-1$ different letters on the board. Also, for a constraint C_1 specified at row R from the left, then that letter has to be at column c_1 or c_2. If it is specified on a row from the right, then the letter has to be either at column c_{N-1} or c_N. If the constraint is applied on a column, the possibilities and logic are identical.

3 Approach

A CSP[2] can be specified by the tuple $\langle V, D, C\rangle$, where V is the set of variables used, D is the variables' domain and C is the set of constraints applied on variables of V.

3.1 Decision Variables

The decision variables list V include every cell of the board, and their domain D is between `0` and `Size-1`, in which `Size` is length of each side of the board (0 equals an empty cell, and numbers between 1 and 27 represent a letter, which is converted later using the `translate` predicate).

3.2 Constraints

The constraints used are, with the domain already applied, the starting and ending of each row/column, i.e, the first element of that row/column looking into the board, starting on that position. Also the `all_distinct` predicate is used to ensure that every row and column have all the necessary elements that belong to the domain.

As specified before, the letter from a certain constraint C at row R from the left can only be at column c_1 or c_2, and from the right it can only be at column c_{N-1} or c_N. To ensure that this happens, the predicate `applyConstraint` uses the `nth1` predicate to get and test the first, second, next to last, and last element. The following figure illustrates the possibilities of A (blue and green cells).

[2] Constraint Satisfaction Problem

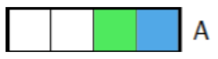

Figure 2. Row Constraint From The Right

Since the row constraint is applied on the right, the letter can only be at column c_{N-1} or c_N. The computer will check the other constraints and test if they all are in a state for finding a solution, taking into consideration the permutations that are being computed on `listsConstraint` predicate.

The solutions are evaluated by `listsConstraint` predicate row by row, i.e, the predicate will try to, by making permutations and applying the constraints using the `applyConstraint` predicate, arrive to a solution in which every row and column must have each letter exactly once.

3.3 Evaluation Function

Since the application was designed only to accept the first solution that finds, there is no evaluation function to estimate the value/cost or goodness of the solution.

However it's easy to determine the number of possibilites of a letter in a certain row (if there are no constraints). Starting from top, let N be the board size and k be the row (1st row is 0), then at the kth row there are $N - k$ possibilities. From this we can conclude that number of possibilities of a letter in a certain row k is given by

$$P(N, k) = N - k$$

When there are constraints, the function that determines all the possible solutions or the "rating"/probability of success given a certain *state* (for example, given the elements of a row) it's quite difficult to calculate because, on this proper case, some situations make the constraints "dependent"on another. The figure below shows the constraints "dependency"in green (because B needs to be the first letter from bottom and A needs to be first letter from left, then A must be at c_2 and B at r_{N-1}).

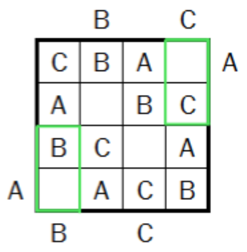

Figure 3. Constraints Dependency

On this simple case and low-size board, the evaluation function would have no trouble evaluating the state of the solution. Also, those simple constraints that have no interference with another constraints can be applied immediately, facilitating the labeling phase. However, with medium or big size boards, and with many constraints "depending"on another, it would cost too much computation time because one constraint (situations in green) can't be applied without taking into consideration the ones which are affected by it. Therefore, it was decided not to implement an evaluation function, giving those premises.

3.4 Search Strategy

The search strategy used is a simple labeling, like `labeling( [], Vars )`. The variables' order is not taken into consideration, only caring if the puzzle is solved or not. This also determines that there is no optimal solution for this problem when there is more than one solution. Since every constraint has to be taken into consideration, the variables order is irrelevant.

4 Solution Presentation

After finding a solution the board is shown to the user using the `drawBoard` predicate. If the player specifies any constraints, they're shown at each side of the board. The application shows the first solution that finds. The below figure illustrates a possible solution for a 5x5 board using `start( [0,2,0,0,0], [0,0,3,1,0], [0,0,1,0,0], [0,0,2,0,0] )`.

```
             C   A
   +---+---+---+---+---+
   |   | B | C | A | D |
   +---+---+---+---+---+
 B | B |   | A | D | C |
   +---+---+---+---+---+
   | C | D |   | B | A | A
   +---+---+---+---+---+
   | A | C | D |   | B |
   +---+---+---+---+---+
   | D | A | B | C |   |
   +---+---+---+---+---+
             B
```

Figure 4. 5x5 Board Possible Solution With Constraints

If the player decides not to specify constraints, then the first solution is immediately shown on the screen, for example, `start( [0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0] )`.

```
+---+---+---+---+
|   | A | B | C |
+---+---+---+---+
| A |   | C | B |
+---+---+---+---+
| B | C |   | A |
+---+---+---+---+
| C | B | A |   |
+---+---+---+---+
```

Figure 5. 4x4 Board Possible Solution Without Constraints

5 Results

After testing different board sizes, the observed results (using the `statistics` predicate and without any constraints) are the ones of the following table.

Board Size	Execution Time (secs)	Time Increase (secs)
5	0.000	-
10	0.015	0.015
15	0.036	0.021
20	0.062	0.026
25	0.187	0.125

Applying Lagrange Interpolation, the interpolation polynomial $T(s)$ that determines the approximate computation time required (according to the observed results and board size) is

$$T(s) = 6.33333 \times 10^{-6}s^4 - 0.000318s^3 + 0.00570167s^2 - 0.03875s + 0.087$$

which is represented by the following plot (the red points indicate the observed results):

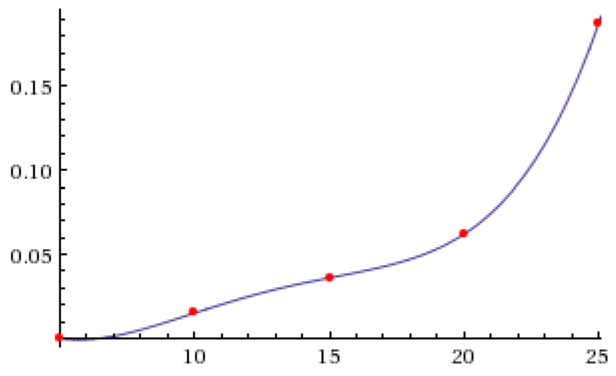

Figure 6. Function T(s)

Judging by the observed results, we can determine that the computation time increases dramatically (for a 50x50 board[3], it was observed that the computer required 3.386 secs to compute a solution). However, the function $T(s)$ gives a misleading computation since $T(50) = 12.237$ secs, which is quite far from what was actually observed.

6 Conclusions and Future Work

This project using CLP helped us to understand that, even if the solution is quite fast in low (2-10) to medium (10-30) sizes, the computation time can increase dramatically when it comes to board sizes greater than it is expected (> 30).

The most important future work is to reduce the computation time (if possible) to at least linear time, even if the game rules only accept letters from the alphabet $\sum = \{A, ..., Z\}$, only being possible to have at maximum a 28x28 board (27 letters and one empty cell per row). In spite of being a bit slow with gigantic boards (out of the game rules), the proposed solution has a great performance by determining a solution for a 25x25 board in only 0.187 secs without any constraints (too many solutions).

This project could be more efficient by using more mathematical models to help excluding some options that don't need to be considered (since there is no solution given some premises). We've gone through Permutation Matrixes and analyzed their properties (eigen values, transpose, determinant, etc.) to permute only solutions without any repetitions on rows/columns. Unfortunately, we couldn't conclude anything useful about those properties to apply on our project.

References

1. Permutation Matrix. Wolfram MathWorld. Retrieved December 12, 2014, from http://mathworld.wolfram.com/PermutationMatrix.html
2. Combinatorial Constraints. SICstus Prolog. Retrieved December 10, 2014, from https://sicstus.sics.se/sicstus/docs/3.12.9/html/sicstus/Combinatorial-Constraints.html
3. Statistics Predicates. SICstus Prolog. Retrieved December 13, 2014, from https://sicstus.sics.se/sicstus/docs/4.2.0/html/sicstus/Statistics-Predicates.html
4. Interpolating Polynomial. Wolfram Alpha. Retrieved December 13, 2014, from http://www.wolframalpha.com/input/?i=interpolating+polynomial+%7B5%2C+0%7D%2C+%7B10%2C+0.015%7D%2C+%7B15%2C+0.036%7D%2C+%7B20%2C+0.062%7D%2C+%7B25%2C+0.187%7D

[3] This size doesn't obey the game rules since the alphabet $\sum = \{A, ..., Z\}$, however, it's a great example of how computation time increases

Annex

```
%Include the needed libraries.
:- use_module( library( lists ) ).
:- use_module( library( clpfd ) ).

createLines( [], _, [] ).
createLines( [BH|BT], Size, Vars ) :-
  length( BH, Size ),
  append( BH, MoreVars, Vars ),
  createLines( BT, Size, MoreVars ).

createBoard( Board, Size, Vars ) :-
  length( Board, Size ),
  createLines( Board, Size, Vars ).

%Applies the given constraints to the given row or column
applyConstraint( List, Size, Start, End ) :-
  nth1( 1, List, Elem1 ), %Get the first element.
  nth1( 2, List, Elem2 ), %Get the second element.
  NextToLast is Size-1,
  nth1( Size, List, ElemLast ), %Get last element.
  nth1( NextToLast, List, ElemNTL ),
  ( Start #= 0 ;
  Elem1 #= Start ;
  ( Elem1 #= 0 , Elem2 #= Start ) ),
  ( End #= 0 ;
  ElemLast #= End ;
  ( ElemLast #= 0 , ElemNTL #= End ) ).

listsConstraint( [], _, [], [], _ ). %Stop condition.
listsConstraint( [BoardH|BoardT], Size, [StartH|StartT],
                 [EndH|EndT], Permutations ) :-
  all_distinct( BoardH ),
  applyConstraint( BoardH, Size, StartH, EndH ),
  listsConstraint(BoardT,Size,StartT,EndT,Permutations).

boardConstraints( Board, Left, Up, Right, Down, Size ) :-
  getPermutations( Size, Permutations ),
  listsConstraint(Board, Size, Left, Right, Permutations),
  transpose( Board, TBoard ),
  listsConstraint( TBoard, Size, Up, Down, Permutations ).

translate( 0, '␣' ).
translate( N, C ) :-
 N < 27,
  NCode is N+64,
  char_code( C, NCode ).
translate( N, NN ) :-
  NN is N-27.

drawLine( [], Separator ) :- %Stop condition.
  write( Separator ).
drawLine( [LineH|LineT], Separator ) :-
  write( Separator ),
  translate( LineH, Letter ), !,
  write( Letter ),
  drawLine( LineT, Separator ).

drawHoriz( 0 ) :-
  write( '+' ), nl.
drawHoriz( N ) :-
  NN is N-1,
  write( '+---' ),
  drawHoriz( NN ).

drawBoardAux( [], _, _, _ ). %Stop condition.
drawBoardAux( [BoardH|BoardT], Size, [LeftH|LeftT],
[RightH|RightT] ) :-
  translate( LeftH, LeftChar ),
  translate( RightH, RightChar ),
  write( '␣' ),
  write( LeftChar ),
  drawLine( BoardH, '␣|␣' ),
  write( RightChar ), nl,
  write( '␣␣␣' ),
  drawHoriz( Size ),
  drawBoardAux( BoardT, Size, LeftT, RightT ).

drawBoard( Board, Size, Left, Up, Right, Down ) :-
  nl,
  write( '␣␣' ),
  drawLine( Up, '␣␣␣' ), nl,
  write( '␣␣␣' ),
  drawHoriz( Size ),
  drawBoardAux( Board, Size, Left, Right ),
  write( '␣␣' ),
  drawLine( Down, '␣␣␣' ), nl,
  nl.
```

```
start :-
  write( 'Usage: start( LeftRestrictions , TopRestrictions ,
RightRestrictions , BottomRestrictions ). ' ), nl,
  write( 'Left and Right restrictions start from the top
and end at the bottom.' ), nl,
  write( 'Top and Bottom restrictions start from the left
and end on the right.' ), nl,
  write( 'Minimum length for all parameters is 2, as a 1x1
Board would always be empty regardless.' ), nl,
  write( 'When something is unspecified , it should be
represented by a zero.' ), nl,
  write( 'Using invalid numbers (such as -1 or numbers
outside of the range) is allowed , but won\'t produce
any results.' ), nl, nl,
  write( 'Example usages:' ), nl,
  write( ' start( [2,2,1], [2,1,1], [1,1,2], [1,2,2] ).'),
  nl,
  write( ' start([0,0,0,1],[0,2,0,3],[1,0,0,0],[2,0,3,0]).
'), nl, nl.
start( _ ) :- start.
start( _, _ ) :- start.
start( _, _, _ ) :- start.

start( Left, Up, Right, Down ) :-
  length( Left, Size ),
  Size > 1,
  length( Up, Size ),
  length( Right, Size ),
  length( Down, Size ),
  createBoard( Board, Size, Vars ),
  Limit is Size-1,
  domain( Vars, 0, Limit ),
  boardConstraints( Board, Left, Up, Right, Down, Size ),
  labeling( [], Vars ),
  drawBoard( Board, Size, Left, Up, Right, Down ).

start( _, _, _, _ ) :-
  write( 'Please ensure that all lists of restrictions
have the same size and that the Board can still have a
solution!' ), nl, nl.

%Pre-made 'queries' to be used with the program.
puzzle0 :-
  start( [0,1], [1,0], [1,1], [0,0] ).
puzzle1 :-
```

```
  start( [2,2,1], [2,1,1], [1,1,2], [1,2,2] ).
puzzle2 :-
  start( [0,0,0,1], [0,2,0,3], [1,0,0,0], [2,0,3,0] ).
puzzle3 :-
  start( [0,0,0,1], [0,0,0,1], [1,0,0,0], [1,0,0,0] ).
puzzle4 :-
  start([0,0,0,1,0],[0,2,0,3,0],[1,0,0,0,0],[2,0,3,0,0]).
puzzle5 :-
  start([4,2,3,1,2],[4,1,2,1,3],[3,1,4,2,1],[0,2,3,4,1]).
```