Diamond Puzzles

Programação em Lógica

João Pedro Dossena - UP201800174

João Basto do Rosário - UP201806334

FEUP-PLOG

Turma 3MIEIC07

Grupo Diamond_4

January 4th, 2020

Porto

Abstract

Our problem was solving a puzzle with diamonds scattered around a MxN rectangular board, in which we had to divide said board in squares, each square containing exactly 1 diamond. The method we used to approach the puzzle was based on 2 premises, the first one is that there are squares that make the top left corner of each square and the second one is that each cell must belong to a square.

The first premise we based on the fact that no 2 cells can be a top left corner of the same square so each square can only have one top corner. The second one we made was that each cell must be located inside a squared shape. When these 2 conditions are met we then imply that if a cell is an upper left corner then the cell is also a square.

Introduction

The objective of this problem was a fitting problem, in which we would have to fit a certain amount of squares on a rectangular board, and in addition, fit the diamonds inside the squares.

In this paper we will report our work, going through the problem description, the approach, the solution presentation, and our experiences and results.

Problem Description

The problem given was a generic MxN rectangular board, in which some cells contained a diamond shape. This board would have to be divided in such a way that each division was shaped like a square, and each square division contained exactly one diamond shape. So the problem could theoretically be subdivided in three subproblems: whether the rectangular board could be divided in as many squares as there are diamonds; where each square would be placed; and what is the length of each square.

Approach

Decision Variables

For our main decision variables we chose the list representing the board, using sometimes a one-dimensional list, and sometimes a two-dimensional one, according to our needs. Contained in this list there are all of the cells of the board, which are the domain variables. Each cell's domain ranges from 1 to the amount of diamonds on the board, which is the intended number of squares. This was chosen because it is easy to represent the board at the end in a clean way, such that “square 1” is a square in which every cell’s value is 1, “square 2” has every cell valued as 2, and so on, and so forth.

Other decision variables that are used are: a boolean flag that is used to decide whether a certain cell is the upper left corner of a square; a counter that marks the amount of equal numbers in a row, or equal lines of the square; a boolean flag that is used to decide when to stop incrementing said counter; a boolean flag that shows if it is a square.

Constraints

The first constraint that we used was gathering a list of the diamond cells’ indexes, and declaring them to be distinct from each other (all_distinct). After that, we use reification in order to establish logical relations between the domain variables to check that a cell is an upper left corner of a square, and the counter and the flags are correct.

Solution Presentation

To present the solution in readable text format, we have a predicate (draw_solve/3) that prints the solution board in which each solution square has its cells filled with its number.

Experiments and Results

We could not experiment anything since we could not finish the problem. It works in a really basic case, and usually gets to the labeling part.

Since we cannot exactly get information, we cannot talk about the results. First and foremost, we understand that our code does not work, probably, due to an error on a domain variable definition.

Conclusions and Future Work

We found this project to be very challenging, because although it does not require many lines of code, it required a lot of thinking about how to restrict variables that do not have values until the very end. Ideally we would have made a program that can solve every solvable diamond puzzle, so that we could experiment with different search methods in the labeling phase.

Referências

https://www.swi-prolog.org/

Anexos

Anexo diamond.pl

```
:- consult('utils.pl').

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

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

createDiamonds([Index|T], Vars, [Diamond| T1]):-
    nth1(Index, Vars, Diamond),
    createDiamonds(T, Vars, T1).


getRest(_, [], _, []).

getRest(DiamondIndexList, [_H|T], Count, Rest):-
    member(Count, DiamondIndexList),
    NewCount is Count + 1,
    getRest(DiamondIndexList, T, NewCount, Rest).

getRest(DiamondIndexList, [H|T], Count, [H|Rest]):-
    \+member(Count, DiamondIndexList),
    NewCount is Count + 1,
    getRest(DiamondIndexList, T, NewCount, Rest).


count([], _, 0).

count([H|T], P, Number):-
    dif(H, P),
    count(T, P, Number).

count([H|T], H, NewNumber):-
    count(T, H, Number),
    NewNumber is Number + 1.

%% "Template" do professor:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%
%% check_cell(Row-Col):-
%%        check_upper_left_corner(Row-Col,IsUL),
```

```
%%        check_square(Row-Col,IsSquare),
%%        IsUL #=> IsSquare.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%
noMore([], _Elem, _ElemIx, _CurrIx).

noMore([_H|T], ElemNotToRepeat, ElemIndex, ElemIndex) :-
    %% write('same element'),nl,
    %% H #= ElemNotToRepeat,
    NewCurrentIndex is ElemIndex + 1,
    noMore(T, ElemNotToRepeat, ElemIndex, NewCurrentIndex).

noMore([H|T], ElemNotToRepeat, ElemIndex, CurrIndex) :-
    %% write('different element'),nl,
    H #\= ElemNotToRepeat,
    %% write('after comparison different'),nl,
    NewCurrentIndex is CurrIndex + 1,
    noMore(T, ElemNotToRepeat, ElemIndex, NewCurrentIndex).


%% check_number_of_equals([], _ULIndex, _CurrentIndex, Total, Total).
%% check_number_of_equals([_H], _ULIndex, _CurrentIndex, Total, Total).
%% check_number_of_equals([_H, NH|T], ULIndex, CurrentIndex, Number,
Total):-
%%  CurrentIndex < ULIndex,
%%  NewCurrentIndex is CurrentIndex + 1,
%%  check_number_of_equals([NH|T], ULIndex, NewCurrentIndex, Number,
Total).
%% check_number_of_equals([H, NH|_T], ULIndex, CurrentIndex, Total,
Total):-
%%  CurrentIndex >= ULIndex,
%%  H #\= NH.
%% check_number_of_equals([H, NH|T], ULIndex, CurrentIndex, Number,
Total):-
%%  CurrentIndex >= ULIndex,
%%  H #= NH,
%%  NewNumber is Number + 1,
```

```
%%  NewCurrentIndex is CurrentIndex + 1,
%%  check_number_of_equals([NH|T], ULIndex, NewCurrentIndex, NewNumber,
Total).


%% check_columns(_List, _NColumns, _Index, _Element, Count, Number, 0):-
%false
%%  Count > Number.
%% check_columns(List, _NColumns, Index, Element, Number, Number, 1):-
%true
%%  element(Index, List, TestElement),
%%  TestElement #\= Element.
%% check_columns(List, NColumns, Index, Element, Count, Number,
IsSquare):-
%%  element(Index, List, TestElement),
%%  TestElement #= Element,
%%  NewCount is Count + 1,
%%  NewIndex is Index + NColumns,
%%  Number < NColumns,
%%  check_columns(List, NColumns, NewIndex, Element, NewCount, Number,
IsSquare).
%% check_columns(List, NColumns, Index, Element, _Count, Number, 0):-
%false
%%  element(Index, List, TestElement),
%%  TestElement #= Element,
%%  %% NewCount is Count + 1,
%%  %% NewIndex is Index + NColumns,
%%  Number >= NColumns.


%% check_square(_List, NRows, NColumns, Index, _IsSquare):-
%%  Index =:= (NRows * NColumns).
%% check_square(List, _NRows, NColumns, Index, IsSquare):-
%%  check_number_of_equals(List, Index, 1, 0, Number),
%%  write('Number: '),
%%  write(Number), nl,
%%  element(Index, List, ElementToCheck),
```

```
%%  check_columns(List, NColumns, Index, ElementToCheck, 0, Number,
IsSquare).


% check_cell(_List, _NRows, _NColumns, FinalIndex, FinalIndex).
%   %% Index > (NRows * NColumns).

% check_cell(List, NRows, NColumns, Index, FinalIndex) :-
% %%    %% Index =< (NRows * NColumns),
%   domain([IsUpperLeftCorner], 0, 1),
%   Index mod NColumns =\= 0,
%   Temp is Index // NRows + 1,
%   Temp < NRows,
%   check_upper_left_corner(List, NRows, NColumns, Index,
IsUpperLeftCorner),
%   %% IsUpperLeftCorner =:= 1,
%   element(Index, List, Elem),
%   RightIndex is Index + 1,
%   element(RightIndex, List, RightElem),
%   BottomIndex is Index + NColumns,
%   element(BottomIndex, List, BottomElem),
%   Elem #= RightElem,
%   Elem #= BottomElem,
% %%    write('Index: '),
% %%    write(Index), nl,
% %%    % write('IsUpperLeftCorner: '),
% %%    % write(IsUpperLeftCorner), nl,
% %%    % check_square %%TODO
% %%    % check_square(List, NRows, NColumns, Index, IsSquare),
% %%    % write('IsSquare: '),
% %%    % write(IsSquare), nl,
% %%    % IsUL #=> IsSquare
% %%    % IsUpperLeftCorner #=> IsSquare,
%   NextIndex is Index + 1,
%   check_cell(List, NRows, NColumns, NextIndex, FinalIndex).
```

```
% check_cell(List, NRows, NColumns, Index, FinalIndex) :- %% Not upperleft
% %%    %% Index < (NRows * NColumns),
%   domain([IsUpperLeftCorner], 0, 1),
%   check_upper_left_corner(List, NRows, NColumns, Index,
IsUpperLeftCorner),
%   %% IsUpperLeftCorner =:= 0,
%   write('Index: '),
%   write(Index), nl,
%   NextIndex is Index + 1,
%   check_cell(List, NRows, NColumns, NextIndex, FinalIndex).

% check_cell(List, NRows, NColumns, Index, FinalIndex) :- %%  UperLeft and
1x1 square
%   domain([IsUpperLeftCorner], 0, 1),
%   check_upper_left_corner(List, NRows, NColumns, Index,
IsUpperLeftCorner),
%   %% IsUpperLeftCorner =:= 1,
%   write('Index of 1x1 square: '),
%   write(Index), nl,
%   element(Index, List, Elem),
%   %% write('before noMore: '),nl,
%   noMore(List, Elem, Index, 1),
%   %% write('after noMore: '),nl,
%   NextIndex is Index + 1,
%   check_cell(List, NRows, NColumns, NextIndex, FinalIndex).




% count_numbers_in_row(List, NRow, NCol, OldElem, Total, Total):-
%   nth1(NRow, List, Row),
%   element(NCol, Row, Elem),
%   Elem #\= OldElem.

count_numbers_in_row(_List, TotalColumns, _NRow, NewNCol, _Elem, 0,
_NotFinished):-
```

```
    NewNCol > TotalColumns.


count_numbers_in_row(List, TotalColumns, NRow, NCol, OldElem, Count, NotFinished):-
    NCol =< TotalColumns,
    nth1(NRow, List, Row),
    nth1(NCol, Row, Elem),
    Count #= CountNext + NotFinished,

    NotFinishedNext #<=> NotFinished #/\ (Elem #= OldElem),

    NewNCol is NCol + 1,
    count_numbers_in_row(List, TotalColumns, NRow, NewNCol, Elem, CountNext, NotFinishedNext).



count_equal_rows(List, TotalRows, TotalColumns, NRow, NCol, _Elem, 0, Sum, _NotFinished):-
    NRow > TotalRows.


count_equal_rows(List, TotalRows, TotalColumns, NRow, NCol, Elem, Count, Sum, NotFinished):-
    NRow =< TotalRows,
    count_numbers_in_row(List, TotalColumns, NRow, NCol, Elem, RowCount, 1),

    Count #= CountNext + NotFinished,
    NotFinishedNext #<=> NotFinished #/\ (RowCount #= Sum),

    NewNRow is NRow + 1,
    count_equal_rows(List, TotalRows, TotalColumns, NewNRow, NCol, Elem, CountNext, Sum, NotFinishedNext).
```

```
check_square(List, Index, NRows, NColumns, IsSquare):-
    NCol is Index mod NColumns + 1,
    NRow is Index // NRows + 1,
    TCol \== 0,
    NCol = TCol,

    nth1(NRow, List, Row),
    element(NCol, Row, Elem),

    count_numbers_in_row(List, NColumns, NRow, NCol, Elem, Sum, 1),


    count_equal_rows(List, NRows, NColumns, NRow, NCol, Elem, Count, Sum,
1),

    Count #= Sum #<=> IsSquare.


check_square(List, Index, NRows, NColumns, IsSquare):-
    TCol is Index mod NColumns,
    NRow is Index // NRows + 1,
    TCol == 0,
    NCol = NColumns,

    nth1(NRow, List, Row),
    element(NCol, Row, Elem),

    count_numbers_in_row(List, NColumns, NRow, NCol, Elem, Sum, 1),

    count_equal_rows(List, NRows, NColumns, NRow, NCol, Elem, Count, Sum,
1),

    Count #= Sum #<=> IsSquare.
```

```
iterateBoard(FlatList, List, NRows, NColumns, Index, FinalIndex):-
    Index =:= FinalIndex - 1.
iterateBoard(FlatList, List, NRows, NColumns, Index, FinalIndex):-
    % write(FinalIndex), nl,
    % write(Index), nl,
    check_upper_left_corner(FlatList, NRows, NColumns, Index,
IsUpperLeftCorner),
    check_square(List, Index, NRows, NColumns, IsSquare),
    IsUpperLeftCorner #=> IsSquare,
    NewIndex is Index + 1,
    iterateBoard(FlatList, List, NRows, NColumns, NewIndex, FinalIndex).




%% SE NÃO ME ENGANO FAZ SENTIDO O PRIMEIRO ELEMENTO SEMPRE SER UM UPPER
LEFT CORNER
% In case the element is the first element, then it logically is an Upper
Left Corner
check_upper_left_corner(_List, _NRows, _NColumns, 1, IsUpperLeftCorner) :-
    IsUpperLeftCorner #= 1. %true

% In case the element is on the first row but not in first column
check_upper_left_corner(List, _NRows, NColumns, Index, IsUpperLeftCorner)
:- %true
    Row is Index // NColumns,
    Row == 1, % Or (Index // NColumns) == 1
    element(Index, List, CurrElement),
    LeftIndex is Index - 1,
    element(LeftIndex, List, LeftElement),
    CurrElement #\= LeftElement,
    IsUpperLeftCorner #= 1. % If they have different values, then the
current element is an Upper Left Corner
% In case the element is not on the first row but is in the first column
check_upper_left_corner(List, NRows, NColumns, Index, IsUpperLeftCorner)
:- %true
```

```
    Column is Index mod NRows,
    Column == 1, % Or (Index mod NRows) == 1
    element(Index, List, CurrElement),
    TopIndex is Index - NColumns,
    element(TopIndex, List, TopElement),
    CurrElement #\= TopElement,
    IsUpperLeftCorner #= 1. % If they have different values, then the
current element is an Upper Left Corner

% In case the element is neither on the first row nor is it in the first
column
check_upper_left_corner(List, _NRows, NColumns, Index, IsUpperLeftCorner)
:- %true
    element(Index, List, CurrElement),
    TopIndex is Index - NColumns,
    element(TopIndex, List, TopElement),
    CurrElement #\= TopElement,
    LeftIndex is Index - 1,
    element(LeftIndex, List, LeftElement),
    CurrElement #\= LeftElement,
    IsUpperLeftCorner #= 1. % If it is different from both the top and the
left, then the current element is an Upper Left Corner

%% SE NÃO ME ENGANO, ANALISEI TODOS OS CASOS VERDADEIROS
% In the other cases it is false
check_upper_left_corner(_List, _NRows, _NColumns, _Index,
IsUpperLeftCorner) :-
    IsUpperLeftCorner #= 0. %false

% DiamondIndexList is a list with the index number of each diamond on the
original problem board

% Examples to copy & paste:
% solve([1, 4, 7, 18, 33, 42, 43, 47, 49], 7, 7, SolutionBoard).

% Main function that solves puzzle
```

```
% solve(+DiamondIndexList, +NumberOfRows, +NumberOfColumns,
-SolutionBoard).
solve(DiamondIndexList, NumberOfRows, NumberOfColumns, Vars) :-
    % Draws Problem Board
    draw(1, NumberOfColumns, NumberOfRows, DiamondIndexList),

    % Timer starts
    statistics(walltime, [Start,_]),

    % Decision Variables
    length(DiamondIndexList, NumberOfDiamonds), % Gets number of diamonds
in problem
    length(SolutionBoard, NumberOfRows),        % Sets numbers of rows to
solution
    build_cols(SolutionBoard, NumberOfColumns), % Sets numbers of columns
to solution
    append(SolutionBoard, Vars),                % Flattens solution
    domain(Vars, 1, NumberOfDiamonds),          % Sets domain of each cell
in solution

    createDiamonds(DiamondIndexList, Vars, DiamondList),
    getRest(DiamondIndexList, Vars, 1, Rest),

    %Criar variaveis que representem os numeros

    % Restrictions
    all_distinct(DiamondList),
    % trace,
    %% write('Before checking cells'), nl,
    FinalIndex is NumberOfRows * NumberOfColumns + 1,
    % check_cell(Vars, NumberOfRows, NumberOfColumns, 1, FinalIndex),
    % trace,
    iterateBoard(Vars, SolutionBoard, NumberOfRows, NumberOfColumns, 1,
FinalIndex),
    % notrace,
    %% write('After checking cells'), nl,
    /*V1,V1,T
```

```
    nth1(1+ColunmNumber, Vars, V1)*/

    % all_squares(Vars),

    % O número de vezes que uma variavel aparece tem de ser um quadrado
perfeito

    % Se aparecer mais que 1 vez então tem de estar todos juntos



    % Labeling
    labeling([], Vars),
    % notrace,
    % findall(Count, (between(1, NumberOfDiamonds, N), count(Vars, N,
Count)), CountList),
    % write(Rest), nl,
    % write(CountList), nl,
    % write(Vars), nl,

    % % Timer ends
    % statistics(walltime, [End,_]),
    % Time is End - Start,
    % format('Duration: ~3d s~n', [Time]),


    % % Draws Problem Board
    draw_solve(NumberOfColumns, Vars, NumberOfColumns).
```

draw.pl

```


draw_puzzle(Largura, 0, L):-
    draw_division(Largura, L), nl.
```

```
draw_puzzle(Largura, Altura, L):- %L são as coordenadas dos diamantes
    draw_division(Largura, L), nl,
    put_code(9482), draw_line(Largura, L), nl,
    NewAltura is Altura - 1,
    draw_puzzle(Largura, NewAltura, L).


draw_line(0, L).

draw_line(Largura, Altura, [[X, Y]|_]):-
    dif(Largura, X),
    dif(Altura, X),
    draw_cell(0),
    NewLargura is Largura - 1,
    draw_line(NewLargura, L).


draw_cell(0):-
    write('  '), put_code(9482).

draw_division(0, L):- put_code(9480).

draw_division(Largura, L):-
    put_code(9480), put_code(9480), put_code(9480),
    NewLargura is Largura - 1,
    draw_division(NewLargura, L).
```

utils.pl

```
build_cols([],_).
build_cols([Line|Ls],N):-
    length(Line,N),
    build_cols(Ls,N).
```

```prolog
draw(Y,_NumberColumns, NumberRows,_):-
    Y > NumberRows.
draw(Y,NumberColumns, NumberRows,Vars):-
    Y =< NumberRows,
    draw_line(Y,1,NumberColumns,Vars),nl,
    Y1 is Y + 1,
    draw(Y1,NumberColumns, NumberRows, Vars).

draw_line(_,X,N,_):-
    X > N.
draw_line(Y,X,N,Vars):-
    X =< N,
    K is (Y - 1)*N + X,
    draw_cell(K,Vars),write(' '),
    X1 is X + 1,
    draw_line(Y,X1,N,Vars).

draw_cell(K,Vars):-
    member(K,Vars),!,
    write('*').
draw_cell(_,_):-
    write('.').

draw_solve(_, [], _).

draw_solve(Ncolumn, [Elem|T], Current):-
    Current =< 1,
    write(Elem), nl,
    draw_solve(Ncolumn, T, Ncolumn).

draw_solve(Ncolumn, [Elem|T], Current):-
    Current > 1,
    write(Elem), write(' '),
    NewCurrent is Current - 1,
    draw_solve(Ncolumn, T, NewCurrent).
```

```
nth11(Y,X,LL,Elem):-
    nth1(Y,LL,L),
    nth1(X,L,Elem).


%% % Counts number of 1's in a binary list with 0's and 1's (aka counts
diamonds on the board)
%% % count_diamonds(+List, -NumberOfDiamondsInList).
%% count_diamonds([], 0).
%% count_diamonds([1|Tail], OneMoreDiamond) :-
%%  count_diamonds(Tail, N),
%%  OneMoreDiamond is N + 1.
%% count_diamonds([0|Tail], NumberOfDiamonds) :-
%%  count_diamonds(Tail, NumberOfDiamonds).

puzzle :-
    generate_random_puzzle(5, 10,10, [L, NRows, NCols]),
    draw(1, NCols, NRows, L).


generate_random_puzzle(NumDiamonds, MaxRows, MaxCols, [L, NRows, NCols])
:-
    random(2, MaxRows, NRows),
    random(2, MaxCols, NCols),
    MaxIndex is NRows * NCols,
    generate_random_diamonds(0, NumDiamonds, MaxIndex, [], L).



generate_random_diamonds(NumDiamonds, NumDiamonds, _MaxIndex, Acc, Acc).
generate_random_diamonds(Index, NumDiamonds, MaxIndex, Acc, List) :-
    random(1, MaxIndex, NewDiamond),
    append([NewDiamond], Acc, NewAcc),
    NewIndex is Index + 1,
```

```
    generate_random_diamonds(NewIndex, NumDiamonds, MaxIndex, NewAcc,
List).
```