Implementação de um solver para o puzzle Square usando programação em lógica com restrições

Catarina Fernandes (up201806610) e Jéssica Nascimento (up201806723)
FEUP-PLOG, Turma 3MIEIC06, Grupo Square_3.

Resumo: Este projeto tem como objetivo aplicar os conhecimentos sobre programação em lógica usando restrições para resolver o puzzle Square. Para isso, foi desenvolvido em SICStus Prolog um programa que resolve um puzzle de dimensões DxD, sendo D uma dimensão variável.

1. Introdução:

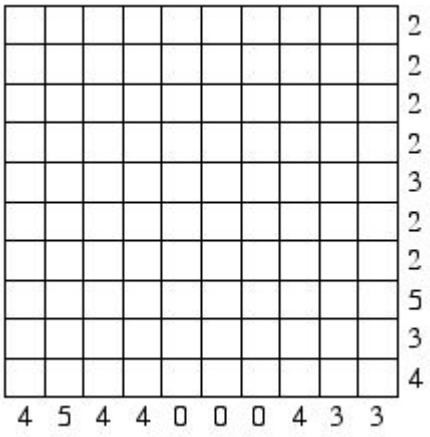
O projeto, desenvolvido no âmbito da unidade curricular de Programação Lógica, teve como objetivo a implementação de um programa em programação em lógica com restrições. O grupo escolheu um problema de decisão, o puzzle Square.

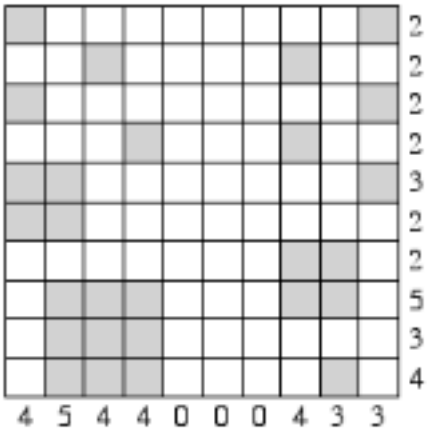
Este artigo está organizado da seguinte forma:

- ➔ **Descrição do problema**: descrição com detalhe do problema de otimização ou decisão em análise.
- ➔ **Abordagem:** descrição da modelação do problema como um PSR, de acordo com:
  - **Variáveis de decisão:** descrição das variáveis de decisão e respetivos domínios.
  - **Restrições:** descrição das restrições rígidas e flexíveis do problema e a sua implementação em SICStus Prolog.
- ➔ **Visualização da solução:** explicação dos predicados que permitem visualizar a solução em modo de texto.
- ➔ **Experiências e resultados**
  - **Análise Dimensional:** exemplos de execução em instâncias do problema com diferentes dimensões e análise dos resultados obtidos.
  - **Estratégias de pesquisa:** apresentação de várias estratégias de pesquisa testadas e comparação com os resultados obtidos.
- ➔ **Conclusões e trabalho futuro:** conclusões retiradas deste projeto e dos resultados obtidos, vantagens e limitações da solução proposta.
- ➔ **Referências:** fontes bibliográficas usadas.
- ➔ Anexo

2. Descrição do problema:

Square é um puzzle que contém uma grelha DxD, sendo D a dimensão da grelha. Contém também um número para cada linha e outro para cada coluna, que representam os números de quadrados que cada linha/coluna têm preenchidos. Para além destas duas restrições, existe uma terceira: os quadrados pintados, têm de fazer parte de um quadrado "mãe" que tem dimensões entre 1 e D, nunca se tocando ou sobrepondo uns aos outros, ou seja tem de ter 1 quadrado de distância entre cada quadrado "mãe".





3. Abordagem:

Para resolver o puzzle optou-se por uma abordagem em que era calculado o número máximo de cantos que era possível ter numa matriz DxD. Essa informação seria usada para gerar 4 listas, uma lista de retângulos, coordenadas x, coordenadas y e tamanhos de quadrados.

Posto isto, seria definido o domínio das listas e gerados os quadrados, usando a lista de retângulos gerada anteriormente. De seguida, seriam aplicadas as restrições de soma das linhas e colunas e feito o labeling.

a. Variáveis de decisão:

O problema recebe duas listas, **Rows** e **Columns**, e, com base nelas, calcula o tamanho da matriz de resolução e coloca esse valor na variável **RowSize**. A partir desse valor é calculado o número de cantos máximos que podem existir:

$$(RowSize \; mod \; 2) \; == \; 0 \; \rightarrow Size \; = \; (RowSize \; / \; 2)^2$$

$$(RowSize \; mod \; 2) \; == \; 1 \; \rightarrow Size \; = \; ((RowSize \; + \; 1) \; / \; 2)^2$$

De seguida criamos as listas auxiliares à aplicação das restrições necessárias. Em *build_lists/10* são criadas 4 listas:

- **NewRectangles:** lista constituída por retângulos:

  [rect(Ax, L1, Ay, L1, a)]

  sendo **Ax** a coordenada x inicial do retângulo e **L1** a dimensão do lado associado, **Ay** a coordenada y inicial do retângulo e **L1** a dimensão do outro lado.

  Aqui é aplicada a restrição do retângulo criado ter o mesmo tamanho em ambos os lados, formando assim apenas quadrados, completando uma das restrições do problema.
- **NewStartX:** lista cujos elementos representam a coordenadas x iniciais dos vários retângulos, ou seja, o Ax de cada quadrado.
- **NewStartY:** lista cujos elementos representam a coordenadas y iniciais dos vários retângulos, ou seja, o Ay de cada quadrado.
- **NewLengths:** lista cujos elementos representam a largura do lado de cada quadrado, ou seja, L1 de cada quadrado.

São então atribuídos os domínios às listas **NewStartX**, **NewStartY** e **NewLengths**, de 0 a **RowSize**.

b. Restrições:

As restrições pedidas são:

1. A soma de uma linha tem que ser igual ao número indicado à sua direita
2. A soma de uma coluna tem que ser igual ao número indicado em baixo
3. As figuras geradas só podem ser quadrados
4. Os quadrados não se podem tocar

Para aplicar as restrições nº3 e nº4 optou-se por usar o predicado *disjoint2/2*. É passada a lista **NewRectangles,** e definida a distância mínima a que cada quadrado tem que estar dos outros.

A lista **NewRectangles** por si só já aplica a restrição nº3, dado que obriga a que os dois lados de cada figura gerada sejam iguais.

```
disjoint2(NewRectangles, [margin(a,a,1,1)])
```

O passo seguinte foi implementar as restrições nº1 e nº2, para isso foi utilizado o predicado *line_constraints/4* e o predicado auxiliar *check _line/4*.

```
line_constraints(_, _, _, []).
line_constraints(Coordenates, Lengths, LineNo, [LineTotal|RestTotals]):-
    check_line(Coordenates, Lengths, LineNo, Counter),
    LineNo2 is LineNo + 1,
    Counter #= LineTotal,
    line_constraints(Coordenates, Lengths, LineNo2, RestTotals).
```

```
check_line([], [], _, 0).
check_line([X|RestX], [L|RestL], LineNo, Counter):-
    LineNo #>= X #/\  LineNo #< (X + L) #<=> B,
    Counter #= Counter2 + (B*L),
    check_line(RestX,RestL, LineNo, Counter2).
```

O predicado *line_constraints/4* é chamado duas vezes, uma para a lista das coordenadas x iniciais e outra para a lista das coordenadas y iniciais.

4. Visualização da Solução:

A visualização da solução problema é feita após o labeling de todas as variáveis na das listas **NewStartX**, **NewStartY**, **NewLengths**.

```
NewStartX: [1,1,2,4,4,5,5,5,6]
NewStartY: [1,6,5,5,6,1,4,5,4]
NewLengths: [3,0,1,0,0,2,0,2,0]
```

É chamado o predicado *convert/5*, que converte as listas numa matriz **Matrix** de dimensão **RowSize** cujos elementos podem ser 1 ou 0:

- 1 caso o quadrado esteja preenchido
- 0 caso esteja vazio.

Um exemplo de como o predicado interpreta as listas é:

- O quadrado cujas coordenadas do canto superior esquerdo são [1, 1] tem 3 quadradinhos de lado, ou seja, vai se prolongar até preencher a secção
  $$1 <= x <= 3 \land 1 <= y <= 3$$
- O quadrado cujas coordenadas do canto superior esquerdo são [2, 5] tem 1 quadradinho de lado, ou seja, vai se prolongar até preencher a secção
  $$x = 2 \land y = 5$$
- O quadrado cujas coordenadas do canto superior esquerdo são [5, 1] tem 2 quadradinhos de lado, ou seja, vai se prolongar até preencher a secção
  $$5 <= x <= 6 \land 1 <= y <= 3$$
- O quadrado cujas coordenadas do canto superior esquerdo são [5, 5] tem 2 quadradinhos de lado, ou seja, vai se prolongar até preencher a secção
  $$5 <= x <= 6 \land 5 <= y <= 6$$

Finalmente, a matriz resultante é imprimida no ecrã, devidamente formatada, através do predicado *displayMatrix/4* e *displayColumns/4*.

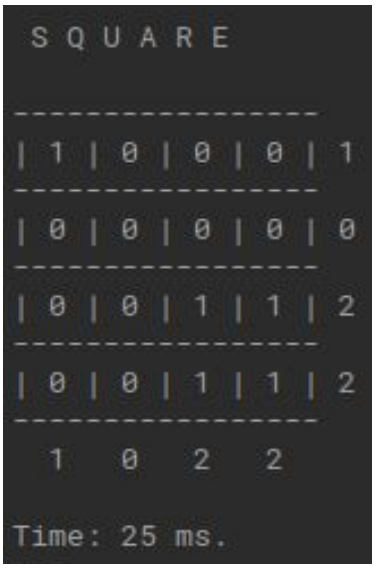
```
S Q U A R E
 _____________________
| 1 | 1 | 1 | 0 | 0 | 0 | 3
 _____________________
| 1 | 1 | 1 | 0 | 1 | 0 | 4
 _____________________
| 1 | 1 | 1 | 0 | 0 | 0 | 3
 _____________________
| 0 | 0 | 0 | 0 | 0 | 0 | 0
 _____________________
| 1 | 1 | 0 | 0 | 1 | 1 | 4
 _____________________
| 1 | 1 | 0 | 0 | 1 | 1 | 4
 _____________________
  5   5   3   0   3   2
```

Rows = [3, 4, 3, 0, 4, 4]
Columns = [5, 5, 3, 0, 3, 2]

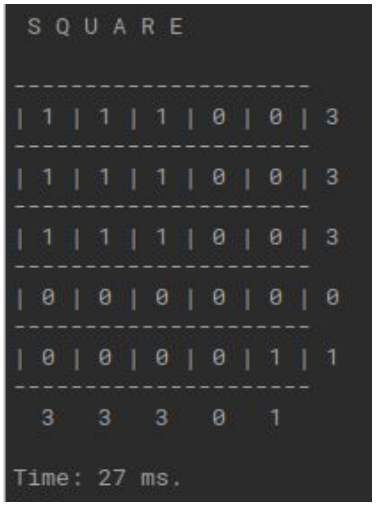
5. Experiências e resultados:

a. Análise Dimensional:

Para analisar se o código funciona para qualquer tipo de dimensão, criamos vários puzzles de várias dimensões. Para efeitos de análise, mostramos aqui 4 dimensões diferentes.



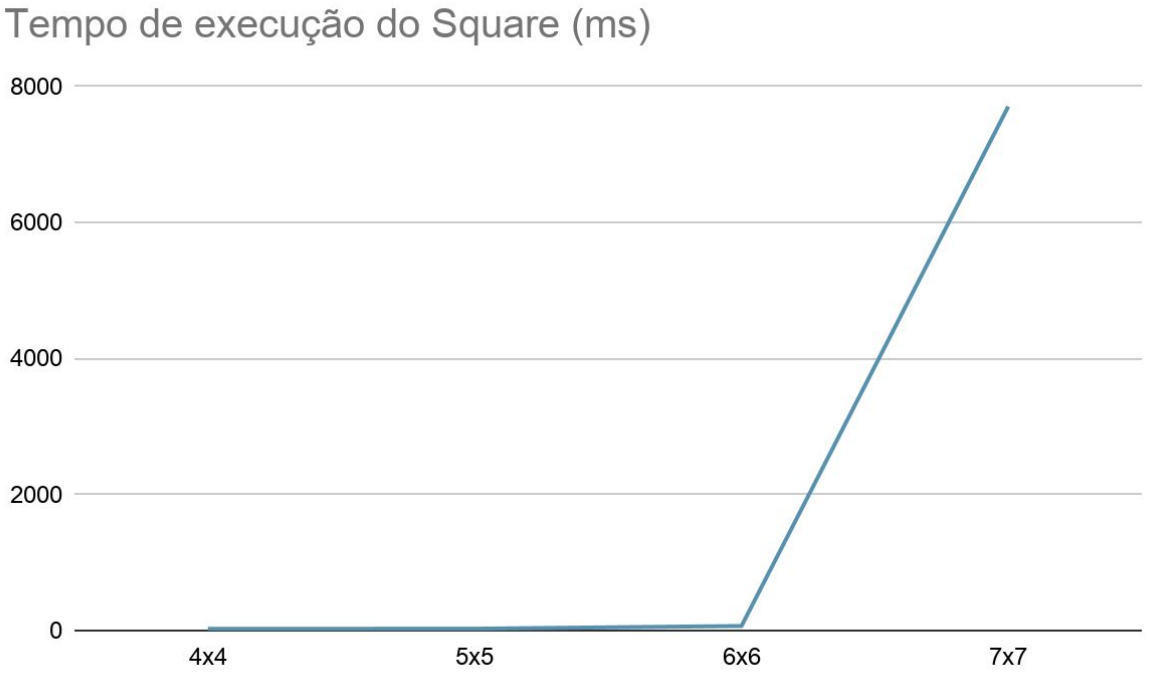
Dimensão: 4x4
Restrições das linhas: [1,0,2,2]
Restrições das colunas: [1,0,2,2]
Tempo: 25ms



Dimensão: 5x5
Restrições das linhas: [3,3,3,0,1]
Restrições das colunas: [3,3,3,0,1]
Tempo: 27 ms

Dimensão: 6x6
Restrições das linhas: [1,1,1,1,1,1]
Restrições das colunas: [1,1,1,1,1,1]
Tempo: 67 ms

Dimensão: 7x7
Restrições das linhas: [1,1,1,1,1,1,1]
Restrições das colunas: [1,1,1,1,1,1,1]
Tempo: 7707 ms



Como podemos verificar, existe um grande aumento no tempo de execução entre o 6x6 e o 7x7.

b. Estratégias de Pesquisa:

Para a estratégia de pesquisa para a resolução deste puzzle foram testadas várias opções. Os testes foram feitos para um puzzle 7x7 e encontram-se no anexo 1. Verificou-se que a melhor opção para o labelling seria o *labeling([max_regret, step, down]*, Vars).

6. Conclusões e trabalho futuro:

Com este trabalho concluímos que o módulo **clpfd** pode ser muito útil, pois consegue aumentar em grande quantidade a eficiência de um solver de problemas de decisão/otimização. Para além disso, percebemos a importância de explorar e experimentar várias heurísticas de pesquisa de forma a encontrar a que fosse mais adequada à nossa resolução, uma vez que a diferença entre elas é notória.

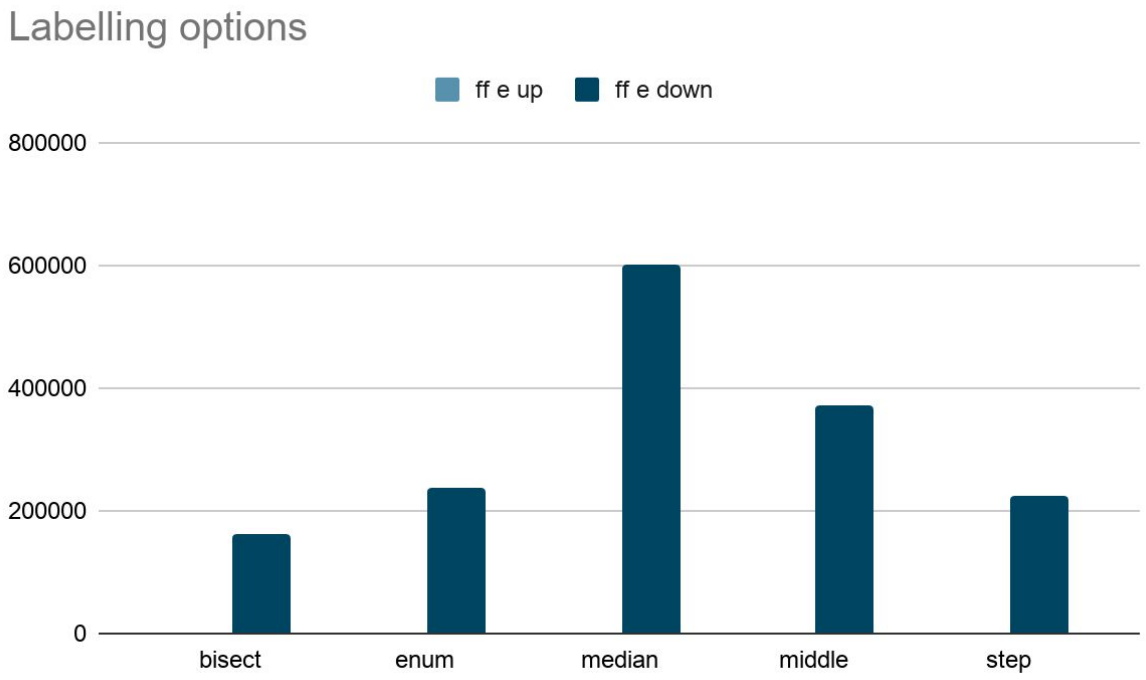
Um trabalho futuro poderá consistir em otimizar a nossa resolução e eliminar possíveis simetrias de forma a diminuir ainda mais o tempo de resolução. Para além disso, poder-se-ia implementar uma variante do puzzle em que há um quadrado marcado por um X, ou seja, não será tido em conta na resolução do problema, devendo o puzzle manter todas as suas restrições na mesma.

Referências:

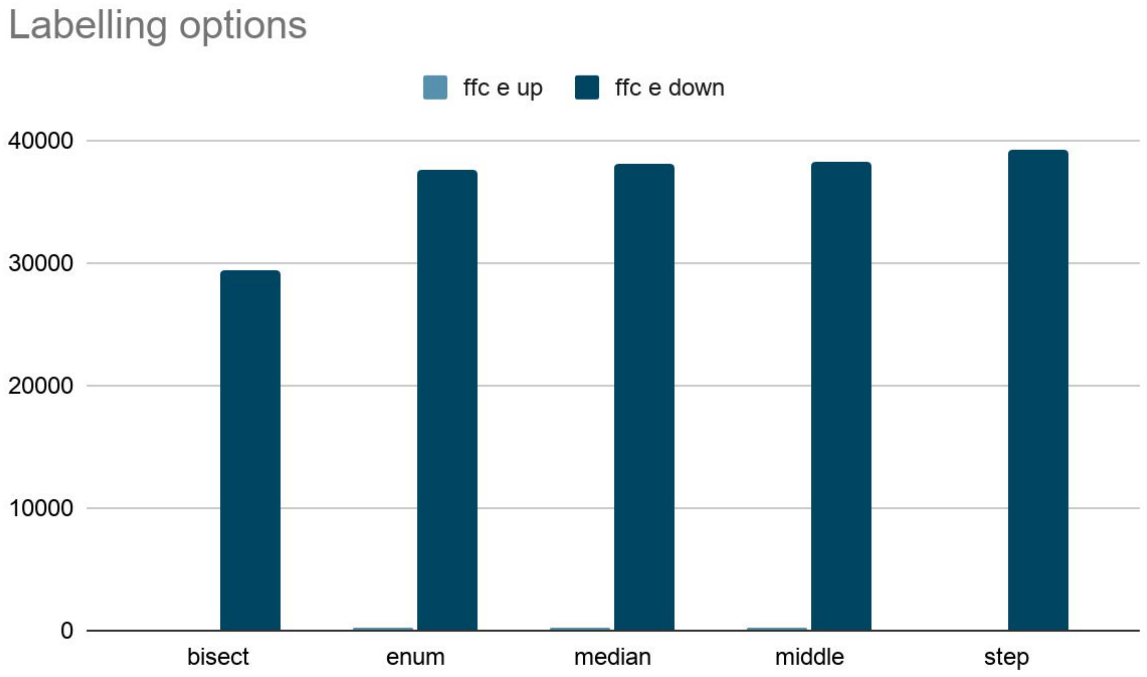
- https://erich-friedman.github.io/puzzle/square/
- https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/Enumeration-Predicates.html

Anexos:

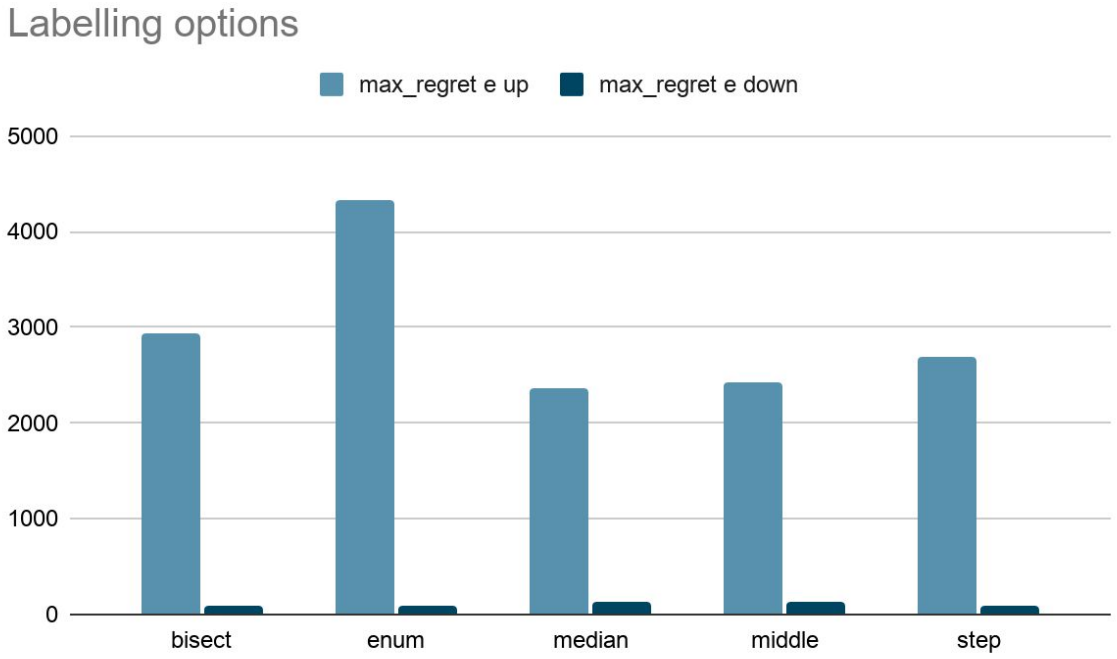
Anexo 1: Labelling options para a estratégia de pesquisa



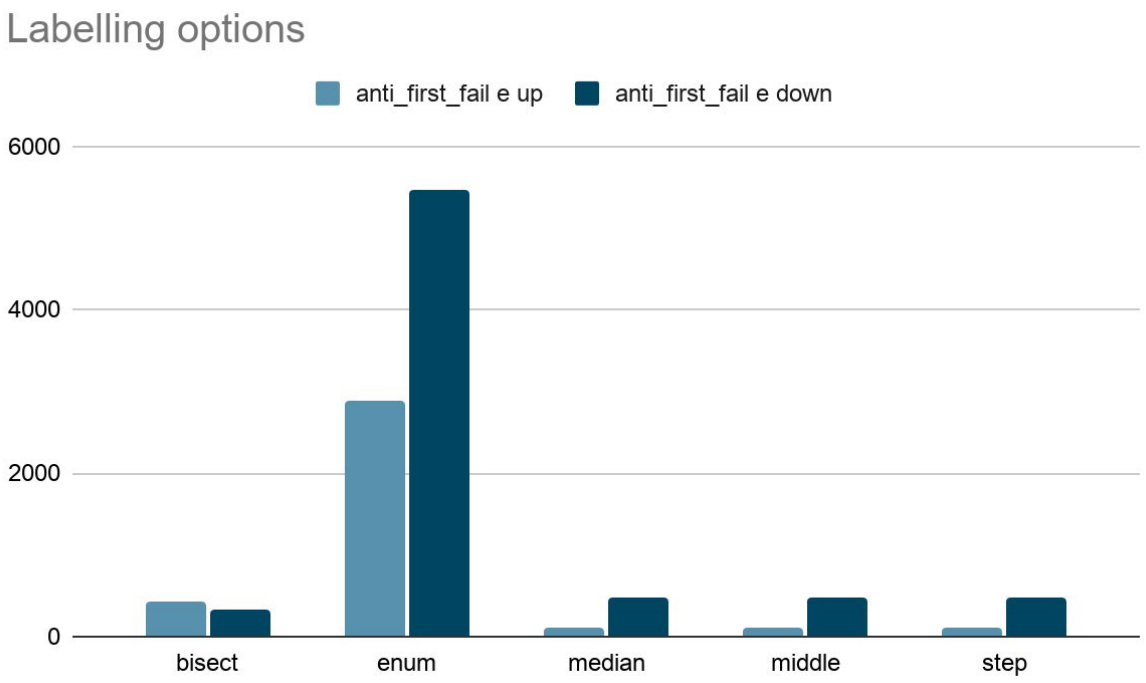
	ff e up	ff e down
bisect	234	162375
enum	688	237875
median	188	601844
middle	203	373265
step	219	223172



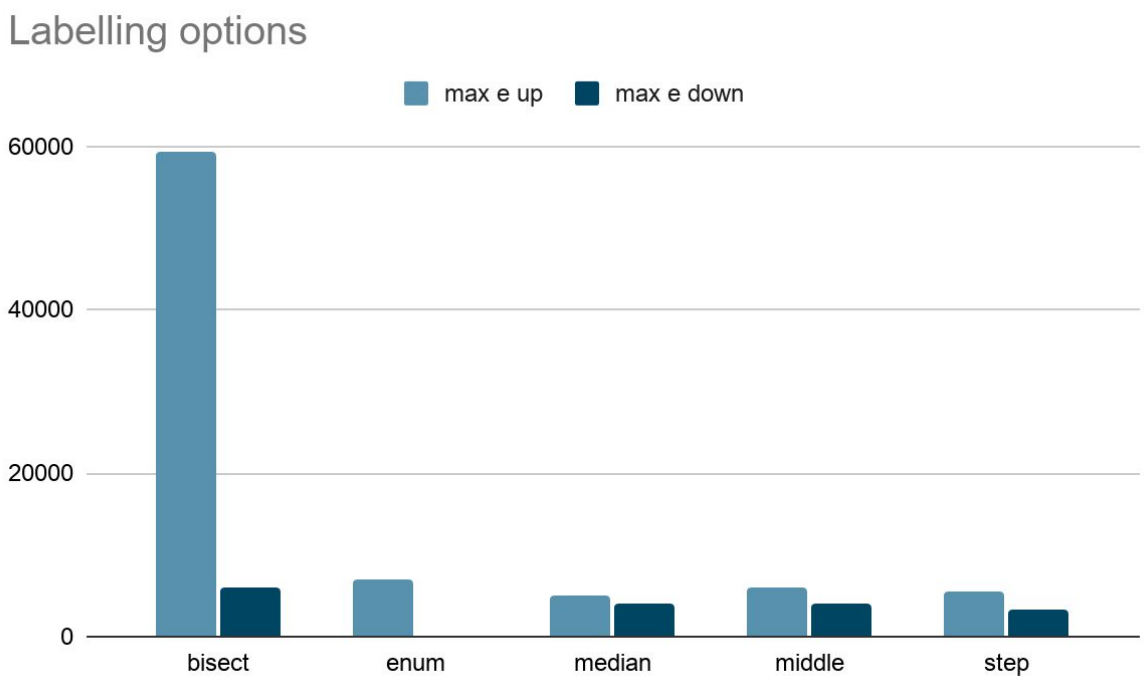
	ffc e up	ffc e down
bisect	117	29473
enum	289	37624
median	158	38105
middle	165	38199
step	146	39303



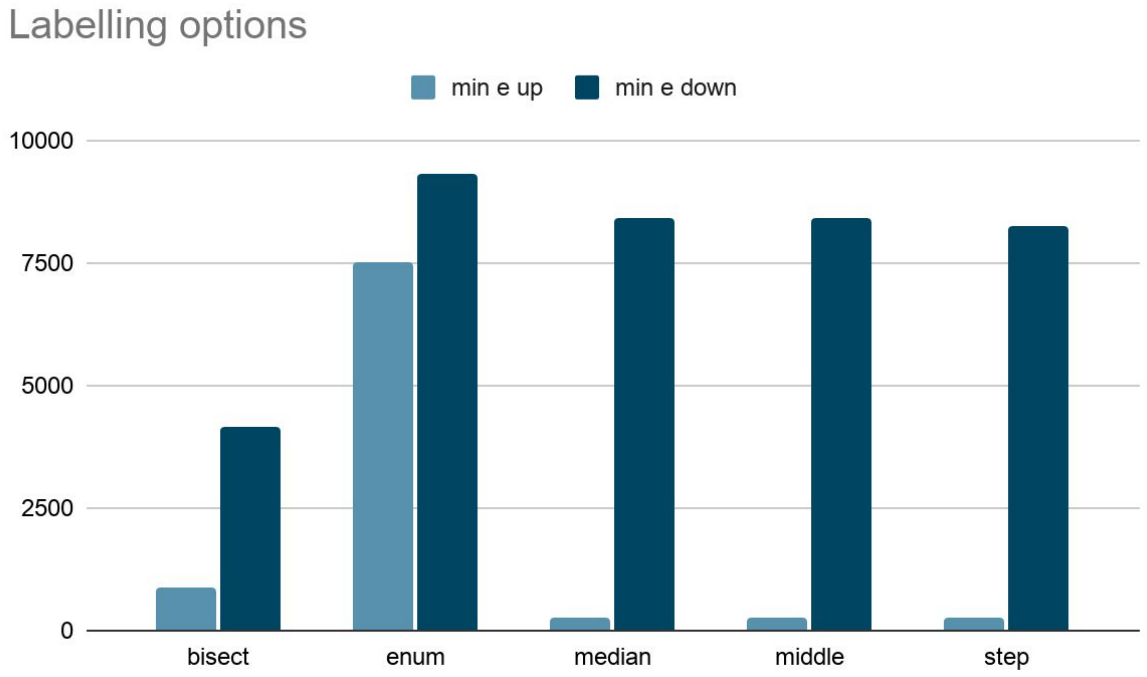
	max_regret e up	max_regret e down
bisect	2941	98
enum	4330	85
median	2369	131
middle	2424	129
step	2686	82



	anti_first_fail e up	anti_first_fail e down
bisect	426	323
enum	2884	5461
median	120	478
middle	101	475
step	117	475



	max e up	max e down
bisect	59436	6124
enum	7010	172
median	5101	4027
middle	6020	4004
step	5513	3413



	min e up	min e down
bisect	891	4140
enum	7521	9322
median	277	8419
middle	279	8413
step	283	8231

Anexo 2: Código fonte

display.pl

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

displayName:-
    nl,write(' S Q U A R E '), nl.

writeRow(Number,Number,Row):-
    nth1(Number,Row,Elem),
    write(Elem),
    write(' |').

writeRow(0,Number,Row):-write('| '),writeRow(1,Number,Row).

writeRow(N,Number,Row):-
    nth1(N,Row,Elem),
    write(Elem),
    write(' | '),
    NewN is N +1,
    writeRow(NewN,Number,Row).

displayColumns(Dimension,Dimension,Columns):-
    nth1(Dimension,Columns,Elem),
    write(Elem).

displayColumns(0,Dimension,Columns):-write('
'),displayColumns(1,Dimension,Columns).

displayColumns(N,Dimension,Columns):-
    nth1(N,Columns,Elem),
    write(Elem),
    write('   '),
    NewN is N +1,
    displayColumns(NewN,Dimension,Columns).


writeLine(0):-write('-').
writeLine(Number):-
    write('----'),
    NewNumber is Number - 1,
    writeLine(NewNumber).


displayMatrix(Matrix,Dimension,Dimension,Rows):-
    writeLine(Dimension),nl,
    nth1(Dimension,Matrix,Row),
    writeRow(0,Dimension,Row),
```

```
    nth1(Dimension,Rows,RowNumber),
    write(' '),write(RowNumber),nl,
    writeLine(Dimension).

displayMatrix(Matrix,Dimension,Number, Rows):-
    writeLine(Dimension),nl,
    nth1(Number,Matrix,Row),
    writeRow(0,Dimension,Row),
    nth1(Number,Rows,RowNumber),
    write(' '),write(RowNumber),nl,
    NewNumber is Number + 1,
    displayMatrix(Matrix,Dimension,NewNumber,Rows).
```

square.pl

```
:- use_module(library(clpfd)).
:- use_module(library(lists)).
:- include('display.pl').

square(Rows, Columns) :-
    displayName,nl,
    statistics(runtime, [T0|_]),

    length(Rows, RowSize),
    get_size(RowSize, Size),

    build_lists(Rectangles, StartX, StartY, Lengths, NewRectangles,
NewStartX, NewStartY, NewLengths, Size, RowSize),

    %Builds domain
    domain(NewStartX, 1, RowSize),
    domain(NewStartY, 1, RowSize),
    domain(NewLengths, 0, RowSize),

    % Restrictions
    disjoint2(NewRectangles, [margin(a,a,1,1)]),
    line_constraints(NewStartX, NewLengths, 1, Rows),
    line_constraints(NewStartY, NewLengths, 1, Columns),

    % Optimization
    orderedSolution(NewStartX, NewStartY),

    % Labeling
    append(NewStartX, NewStartY, V),
    append(V, NewLengths, Vars), !,
    labeling([max_regret, step, down], Vars),
    % write('NewStartX: '), write(NewStartX), nl,
```

```
    % write('NewStartY: '), write(NewStartY), nl,
    % write('NewLengths: '), write(NewLengths), nl,

    % Converts to a 0/1 matrix
    convert(NewStartX, NewStartY, NewLengths, RowSize, Matrix),

    % Display
    displayMatrix(Matrix, RowSize, 1, Rows),nl,
    displayColumns(0,RowSize,Columns),!,nl,
    statistics(runtime, [T1|_]),
    T is T1 - T0,nl,
    format('Time: ~d ms.~n', [T]).

%----------------------------------------------------------------------


% Removes symetries
orderedSolution([_], [_]).
orderedSolution([X1,X2|X], [Y1,Y2|Y]):-
    (X1 #= X2 #/\ Y1 #< Y2) #\/ X1 #< X2,
    orderedSolution([X2|X],[Y2|Y]),!.


% -------------------- Converts Lists to Matrix ----------------------
% filter_lists auxiliar function
filter_lists_Aux(StartX, StartY, Lengths, 1, StartXFiltered, StartYFiltered,
LengthsFiltered) :-
    nth1(1, Lengths, Elem),
    Elem > 0,
    append([], [Elem], LengthsFiltered),
    nth1(1, StartX, ElemX),
    nth1(1, StartY, ElemY),
    append([], [ElemX], StartXFiltered),
    append([], [ElemY], StartYFiltered).
filter_lists_Aux(StartX, StartY, Lengths, 1, StartXFiltered, StartYFiltered,
LengthsFiltered) :-
    nth1(NewLengthsSize, Lengths, Elem),
    Elem == 0,
    StartXFiltered = [],
    StartYFiltered = [],
    LengthsFiltered = [].
filter_lists_Aux(StartX, StartY, Lengths, LengthsSize, NewStartXFiltered,
NewStartYFiltered, NewLengthsFiltered) :-
    NewLengthsSize is LengthsSize - 1,
    nth1(LengthsSize, Lengths, Elem),
    Elem > 0,
    filter_lists_Aux(StartX, StartY, Lengths, NewLengthsSize,
StartXFiltered, StartYFiltered, LengthsFiltered),
```

```
    append(LengthsFiltered, [Elem], NewLengthsFiltered),
    nth1(LengthsSize, StartX, ElemX),
    nth1(LengthsSize, StartY, ElemY),
    append(StartXFiltered, [ElemX], NewStartXFiltered),
    append(StartYFiltered, [ElemY], NewStartYFiltered).
filter_lists_Aux(StartX, StartY, Lengths, LengthsSize, NewStartXFiltered,
NewStartYFiltered, NewLengthsFiltered) :-
    NewLengthsSize is LengthsSize - 1,
    nth1(LengthsSize, Lengths, Elem),
    Elem == 0,
    filter_lists_Aux(StartX, StartY, Lengths, NewLengthsSize,
NewStartXFiltered, NewStartYFiltered, NewLengthsFiltered).


% Removes list elements where Length is 0 and puts them in StartXFiltered,
StartYFiltered and LengthsFiltered
filter_lists(StartX, StartY, Lengths, StartXFiltered, StartYFiltered,
LengthsFiltered) :-
    length(Lengths, LengthsSize),
    filter_lists_Aux(StartX, StartY, Lengths, LengthsSize, StartXFiltered,
StartYFiltered, LengthsFiltered).


% Builds a RowSize x RowSize empty matrix
build_matrix(RowSize, RowSize, [H | []]) :-
    length(List, RowSize),
    append([], List, H).
build_matrix(RowSize, ColumnSize, [H | T]) :-
    NewColumnSize is ColumnSize + 1,
    build_matrix(RowSize, NewColumnSize, T),
    length(List, RowSize),
    append([], List, H).


% Fills row with correpondent 1s
fill_row(StartY, 1, Max, Row) :-
    nth1(StartY, Row, Elem),
    Elem = 1.
fill_row(StartY, Size, SizeConst, Row) :-
    NewStartY is StartY + 1,
    NewSize is Size - 1,
    fill_row(NewStartY, NewSize, SizeConst, Row),
    nth1(StartY, Row, Elem),
    Elem = 1.


% fill_matrix auxiliar function, fills matrix with correspondent 1s
fill_aux(StartX, StartY, 1, SizeConst, Matrix) :-
```

```
    nth1(StartX, Matrix, Row),
    Max is StartY + 1,
    fill_row(StartY, SizeConst, Max, Row).
fill_aux(StartX, StartY, Size, SizeConst, Matrix) :-
    NewStartX is StartX + 1,
    NewSize is Size -1,
    fill_aux(NewStartX, StartY, NewSize, SizeConst, Matrix),
    nth1(StartX, Matrix, Row),
    fill_row(StartY, SizeConst, SizeConst, Row).

% fills matrix with correspondent 1s
fill_matrix(StartXFiltered, StartYFiltered, LengthsFiltered, 1, Matrix,
FilledMatrix) :-
    nth1(1, StartXFiltered, StartXNumber),
    nth1(1, StartYFiltered, StartYNumber),
    nth1(StartXNumber, Matrix, Row),
    nth1(StartYNumber, Row, Elem),
    Elem = 1,
    nth1(1, LengthsFiltered, Size),
    fill_aux(StartXNumber, StartYNumber, Size, Size, Matrix).
fill_matrix(StartXFiltered, StartYFiltered, LengthsFiltered, AuxSize,
Matrix, FilledMatrix) :-
    NewAuxSize is AuxSize - 1,
    fill_matrix(StartXFiltered, StartYFiltered, LengthsFiltered, NewAuxSize,
Matrix, FilledMatrix),
    nth1(AuxSize, StartXFiltered, StartXNumber),
    nth1(AuxSize, StartYFiltered, StartYNumber),
    nth1(StartXNumber, Matrix, Row),
    nth1(StartYNumber, Row, Elem),
    Elem = 1,
    nth1(AuxSize, LengthsFiltered, Size),
    fill_aux(StartXNumber, StartYNumber, Size, Size, Matrix).

% complete_matrix auxiliar function, fills leftover matrix cells with 0s
complete_aux(Row, 1) :-
    nth1(1, Row, Elem),
    Elem \== 1,
    Elem = 0.
complete_aux(Row, 1).
complete_aux(Row, RowSize) :-
    nth1(RowSize, Row, Elem),
    Elem \== 1,
    Elem = 0,
    NewRowSize is RowSize - 1,
    complete_aux(Row, NewRowSize).
complete_aux(Row, RowSize) :-
    nth1(RowSize, Row, Elem),
    Elem == 1,
```

```
    NewRowSize is RowSize - 1,
    complete_aux(Row, NewRowSize).

% Fills leftover matrix cells with 0s
complete_matrix(Matrix, 1, RowSize) :-
    nth1(1, Matrix, Row),
    complete_aux(Row, RowSize).
complete_matrix(Matrix, RowSize, ConstRowSize) :-
    NewRowSize is RowSize - 1,
    complete_matrix(Matrix, NewRowSize, ConstRowSize),
    nth1(RowSize, Matrix, Row),
    complete_aux(Row, ConstRowSize).


% Converts StartX, StartY, Lengths to a RowSize x RowSize matrix
convert(StartX, StartY, Lengths, RowSize, Matrix) :-
    filter_lists(StartX, StartY, Lengths, StartXFiltered, StartYFiltered,
LengthsFiltered),
    build_matrix(RowSize, 1, Matrix),
    length(StartXFiltered, AuxSize),
    fill_matrix(StartXFiltered, StartYFiltered, LengthsFiltered, AuxSize,
Matrix, FilledMatrix),
    complete_matrix(Matrix, RowSize, RowSize).
% ------------------------------------------------------------------------


% Builds NewStartX, NewStartY, NewLength lists with corresponding
restrictions and according to the Size of the Row/Column lists passed in
FixedSize variable
build_lists(Rectangles, StartX, StartY, Lengths, NewRectangles, NewStartX,
NewStartY, NewLengths, 1, FixedSize) :-
    NewRectangles = [rect(Ax, L1, Ay, L1, a)],
    NewStartX = [Ax],
    NewStartY = [Ay],
    NewLengths = [L1],
    Ax + L1 #=< (FixedSize+1),
    Ay + L1 #=< (FixedSize+1).
build_lists(Rectangles, StartX, StartY, Lengths, ResultRectangles,
ResultStartX, ResultStartY, ResultLengths, Size, FixedSize) :-
    NewSize is Size - 1,
    build_lists(Rectangles, StartX, StartY, Lengths, NewRectangles,
NewStartX, NewStartY, NewLengths, NewSize, FixedSize),
    append(NewRectangles, [rect(Ax, L1, Ay, L1, a)], ResultRectangles),
    append(NewStartX, [Ax], ResultStartX),
    append(NewStartY, [Ay], ResultStartY),
    append(NewLengths, [L1], ResultLengths),
    Ax + L1 #=< (FixedSize+1),
    Ay + L1 #=< (FixedSize+1).
```

```
% Calculates the maximum number of corners there can be in a RowSize x
RowSize matrix
get_size(RowSize, Size) :-
    Flag is RowSize mod 2,
    Flag == 0,
    Size is (RowSize // 2) * (RowSize // 2).
get_size(RowSize, Size) :-
    Flag is RowSize mod 2,
    Flag == 1,
    Size is ((RowSize + 1)//2) * ((RowSize + 1)//2).


% line_constraints(CoordenateList, LengthsList, LineNumber, Constraint)
% Enforce row and column sum according to the list they receive
line_constraints(_, _, _, []).
line_constraints(Coordenates, Lengths, LineNo, [LineTotal|RestTotals]):-
    check_line(Coordenates, Lengths, LineNo, Counter),
    LineNo2 is LineNo + 1,
    Counter #= LineTotal,
    line_constraints(Coordenates, Lengths, LineNo2, RestTotals).

% line_constraints auxiliar function
check_line([], [], _, 0).
check_line([X|RestX], [L|RestL], LineNo, Counter):-
    LineNo #>= X #/\  LineNo #< (X + L) #<=> B,
    Counter #= Counter2 + (B*L),
    check_line(RestX,RestL, LineNo, Counter2).
```