

# Yin Yang - Problema de Decisão

## Resolução de Problema de Decisão usando Programação em Lógica com Restrições

José Aleixo Cruz      José Carlos Vieira

*Faculdade de Engenharia da Universidade do Porto*

Dezembro 2016

## Resumo

Neste artigo é abordada a resolução de um puzzle proposto, recorrendo a *Constraints Logic Programming in Finite Domains* (CLP-FD) utilizando Prolog. É referida a descrição do problema bem como os passos tomados para obter a sua solução.

## 1 Introdução

O objectivo deste trabalho era implementar a resolução de um problema de decisão ou de optimização. De entre as escolhas possíveis, o grupo optou pelo puzzle "Yin Yang", que consiste num tabuleiro quadrado onde existem dois tipos de peças: pretas e brancas. As peças da mesma cor têm de estar todas conectadas, verticalmente e horizontalmente, e também não podem existir grupos de 2x2 peças somente com a mesma cor.

## 2 Descrição do Problema

O puzzle "Yin Yang" é jogado num tabuleiro quadrado (de **2x2** ou maior). Existem dois tipos de peças, brancas e pretas. O objectivo do jogo consiste em manter um caminho ligado entre as peças da mesma cor, ou seja, todas as peças brancas têm de estar conectadas a pelo menos uma outra peça

branca, formando um caminho navegável entre elas (aplica-se o mesmo princípio para as peças pretas). Também não podem existir peças numa área **2x2** com apenas uma cor, forçando então que numa área com quatro quadrados (**2x2**) existam sempre peças de diferentes cores.

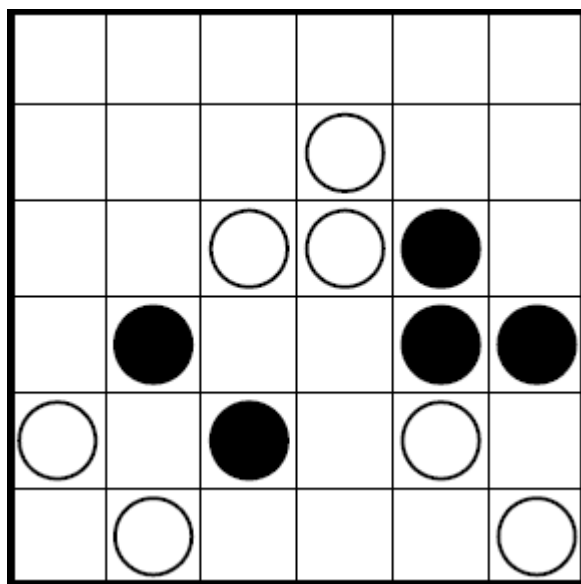


Figura 1: Tabuleiro inicial

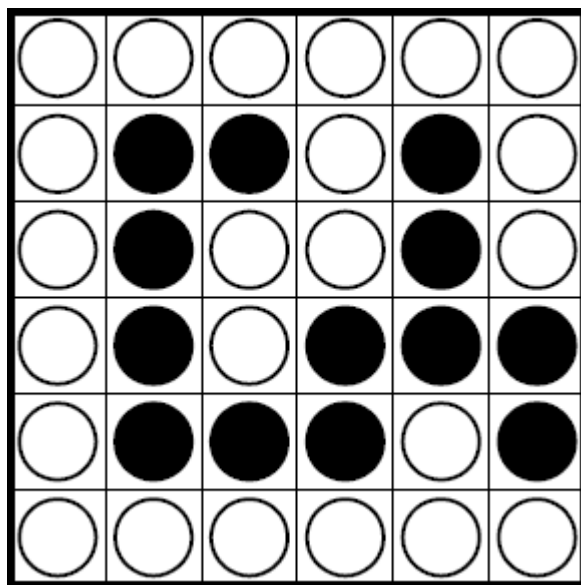


Figura 2: Tabuleiro final

### 3 Abordagem

O tabuleiro é representado como uma lista, onde cada elemento da lista é uma peça branca ou preta.

Considere-se este exemplo ate ao final do artigo, um tabuleiro **4x4**, onde **N** é o numero de linhas/colunas do tabuleiro (neste caso 4).

A representação em prolog para esse tabuleiro é:

```
test_board_4x4([
    A1, A2, A3, A4,
    B1, B2, B3, B4,
    C1, C2, C3, C4,
    D1, D2, D3, D4
]).
```

### 3.1 Variáveis de Decisão

Para resolver o problema, é criada uma lista com dimensão  $N$ , em que  $N$  é o número de casas do tabuleiro quadrangular. Cada elemento dessa lista é um inteiro que identificará se a peça nessa posição é branca ou preta. O domínio dessa lista é, então, um conjunto de quaisquer dois inteiros escolhidos para diferenciar a cor das peças. Como o problema é simétrico, um valor pode representar tanto a cor branca como a cor preta, sendo que o outro valor representará necessariamente a cor oposta. Na nossa abordagem usamos os valores binários **0** e **1**.

### 3.2 Restrições

Para este problema foi proposto encontrar-se uma solução recorrendo unicamente à aplicação de restrições e não à técnica de *geração e teste*. No entanto, após vasta pesquisa e tentativa, não conseguimos formular nenhum conjunto de restrições capazes de satisfazer **completamente o problema**. Apenas conseguimos elaborar restrições ou com âmbito maior ou com âmbito menor do que o objetivo.

#### 3.2.1 Restrições folgadas

Há duas restrições implícitas no *puzzle*:

1. Todas as peças devem estar ligadas a pelo menos uma outra peça da mesma cor;
2. Nenhum quadrado 2x2 deve ter todas as peças da mesma cor.

Estas restrições são as únicas presentes na versão relaxada (menos abrangente) do nosso algoritmo e são aplicadas, no caso geral, da seguinte forma:

```
(
  CurrElem #= ElemRight
  #\ / CurrElem #= ElemBelow
),

nvalue(2, [CurrElem, ElemRight, ElemBelow, ElemSE])
```

Já que o tabuleiro é percorrido da esquerda para a direita e de cima para baixo, a propagação permite-nos restringir o valor apenas da célula abaixo e à direita, evitando condições redundantes.

### 3.2.2 Restrições específicas

Além das restrições menos abrangentes, estabelecemos também que tanto o conjunto de peças brancas como o conjunto de peças pretas devem formar um **caminho de Euler**. Para tal, adicionamos a restrição de que **duas e apenas duas** peças devem ter um número ímpar de peças adjacentes, tal como nos teoremas de Euler.

```
setEulerPathConstraints(Degrees0, Degrees1):-  
    %% apply Euler path to 0s  
    count(1, Degrees0, #=, CountDegree1of0),  
    count(3, Degrees0, #=, CountDegree3of0),  
  
    count(0, Degrees0, #=, 0),  
  
    CountDegree1of0 + CountDegree3of0 #= 2,  
  
    %% apply Euler path to 1s  
    count(1, Degrees1, #=, CountDegree1of1),  
    count(3, Degrees1, #=, CountDegree3of1),  
  
    count(0, Degrees1, #=, 0),  
  
    CountDegree1of1 + CountDegree3of1 #= 2.
```

Aqui, *Degrees0* e *Degrees1* são listas cujos elementos representam os graus das peças atribuídas com o valor 0 ou 1 respetivamente. Isto é, o número de peças iguais que se encontra adjacente a essa peça. Como o importante é analisar os valores em si, não importa nem a ordem nem a que peça corresponde cada elemento.

### 3.3 Estratégia de Pesquisa

A aplicação das restrições percorre a lista que representa as peças do tabuleiro. O primeiro elemento da lista corresponde à peça que está no canto superior esquerdo e o último elemento à peça do canto inferior direito do tabuleiro. Se o tabuleiro tiver como dimensão  $N \times N$ , os primeiros  $N$  elementos representam a primeira linha, os segundos  $N$  elementos a segunda linha, etc..

## 4 Visualização da solução

A nossa função de visualização descreve como seria o tabuleiro "solução" quando representado pelas variáveis de domínio escolhidas para cada peça. Por exemplo, um *output* de uma solução num tabuleiro 5x5, cujo domínio seja  $\{0,1\}$ , será o seguinte:

```
0 0 0 0 0
0 1 0 1 0
0 1 0 1 0
0 1 1 1 0
0 0 0 0 0
```

## 5 Resultados

Ao analisar o desempenho das nossas duas versões, compreendemos que a restrição mais específica causa um aumento exponencial na computação da solução, que aumenta com a dimensão do tabuleiro.

Medimos e comparámos a quantidade de *backtrackings* efetuados. Excluimos a medida de tempo das nossas observações, por causa das diferenças entre processadores.

Tabela 1: Versão relaxada	
Dimensão (NxN)	<i>Backtracks</i> (moda)
2x2	$0 \pm 0$
3x3	$0 \pm 0$
6x6	$3 \pm 1$
8x8	$7 \pm 1$

Tabela 2: Versão restrita	
Dimensão (NxN)	<i>Backtracks</i> (moda)
2x2	$9 \pm 2$
3x3	$22 \pm 5$
5x5	89819 (apenas uma vez)
6x6	não calculado

## 6 Conclusões e Trabalho Futuro

Na abordagem a problemas em Prolog podem ser tomadas duas rotas principais: **propagação** ou **geração e teste**. A programação lógica através de restrições tira proveito do efeito de propagação para obter soluções de forma mais rápida e eficiente que a geração e teste, poupando recursos computacionais.

Assim, antes de resolver um problema, é importante verificar se a situação pode ser eficazmente traduzida num conjunto de restrições, de modo a aplicar um métodos de CLP.

A implementação de um maior número de restrições e/ou aplicação de uma estratégia diferente poderia fornecer um leque mais restringido e com um maior número de soluções do que é possível atualmente.

## Referências

- [1] How can we tell if a graph has an euler path or circuit? <http://www.ctl.ua.edu/math103/euler/howcanwe.htm>, dec 2016.
- [2] Siscstus prolog clp fd library. [https://sicstus.sics.se/sicstus/docs/4.1.0/html/sicstus/lib\\_002dclpfd.html](https://sicstus.sics.se/sicstus/docs/4.1.0/html/sicstus/lib_002dclpfd.html), dec 2016.

## 7 Anexos

`yin_yang_relaxed.pl`

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

/*
0 --> black
1 --> white

LastEvaluatorNumber --> last column of penultima line
*/

test_board_4x4([
    A1, A2, A3, A4,
    B1, B2, B3, B4,
    C1, C2, C3, C4,
```

```

        D1, D2, D3, D4
    ]).

test_board_3x3([
    A1, A2, A3,
    B1, B2, B3,
    C1, C2, C3
]).

test_board_2x2([
    A1, A2,
    B1, B2
]).

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

yin_yang_auto(Board):-
    test_board_4x4(Board),
    length(Board, Length),
    LengthRowAux is sqrt(Length),
    LengthRow is round(LengthRowAux),

    domain(Board, 0, 1),
    setConstrains(Board, 1, Length, LengthRow),
    reset_timer,
    labeling([], Board),
    print_time,
    fd_statistics,

    display_board(Board, LengthRow, 1).

yin_yang_manual(Board, Length):-
    BoardLength is Length * Length,

    length(Board, BoardLength),
    LengthRowAux is sqrt(BoardLength),
    LengthRow is round(LengthRowAux),

    domain(Board, 0, 1),
    setConstrains(Board, 1, BoardLength, LengthRow),
    reset_timer,
    labeling([], Board),

```



```

print_time,
fd_statistics,

display_board(Board, LengthRow, 1).

% cell that ends predicate. Last cell in the board
setConstrains(Board, Length, Length, LengthRow).
setConstrains(Board, LengthRow, Length, LengthRow):-
    % cell that appears in the first row, last column
    NewIndex is LengthRow + 1,

    LeftIndex is LengthRow - 1,
    BelowIndex is LengthRow + LengthRow,

    element(LengthRow, Board, CurrElem),
    element(LeftIndex, Board, LeftElem),
    element(BelowIndex, Board, BelowElem),

    (
        CurrElem #= LeftElem
        #\ CurrElem #= BelowElem
    ),
    setConstrains(Board, NewIndex, Length, LengthRow).
setConstrains(Board, Index, Length, LengthRow):-
    % cell that appears in the last row, first column
    LastRowFirstColumn is Length - LengthRow + 1,
    Index == LastRowFirstColumn,
    NewIndex is Index + 1,

    UpperIndex is Index - LengthRow,
    RightIndex is Index + 1,

    element(Index, Board, CurrElem),
    element(UpperIndex, Board, UpperElem),
    element(RightIndex, Board, RightElem),

    (
        CurrElem #= UpperElem
        #\ CurrElem #= RightElem
    ),
    setConstrains(Board, NewIndex, Length, LengthRow).
setConstrains(Board, Index, Length, LengthRow):-
    % cells between first and last line in the last column (excluded)
    0 == mod(Index, LengthRow),
    NewIndex is Index + 1,

```

```

UpperIndex is Index - LengthRow,
BelowIndex is Index + LengthRow,

element(Index, Board, CurrElem),
element(UpperIndex, Board, UpperElem),
element(BelowIndex, Board, BelowElem),

(
    CurrElem #= UpperElem
    #\ CurrElem #= BelowElem
),
setConstrains(Board, NewIndex, Length, LengthRow).
setConstrains(Board, Index, Length, LengthRow):-
    % cells in the last row between first and last columns (excluded)
    LastRowIndex is Length - LengthRow,
    Index > LastRowIndex,
    NewIndex is Index + 1,

    LeftIndex is Index - 1,
    RightIndex is Index + 1,

    element(Index, Board, CurrElem),
    element(LeftIndex, Board, LeftElem),
    element(RightIndex, Board, RightElem),

    (
        CurrElem #= LeftElem
        #\ CurrElem #= RightElem
    ),
    setConstrains(Board, NewIndex, Length, LengthRow).
setConstrains(Board, Index, Length, LengthRow):-
    % intermediate board cells
    element(Index, Board, CurrElem),
    NextIndex is Index + 1,
    element(NextIndex, Board, ElemRight),
    NextRow is Index + LengthRow,
    element(NextRow, Board, ElemBelow),
    NextRowPlus is NextRow + 1,
    element(NextRowPlus, Board, ElemSE),

    (
        CurrElem #= ElemRight
        #\ CurrElem #= ElemBelow
    ),

    nvalue(2, [CurrElem, ElemRight, ElemBelow, ElemSE]),

```

```

        NewIndex is Index + 1,
        setConstrains(Board, NewIndex, Length, LengthRow).

display_board([], _, _).
display_board([X|Xs], LengthRow, Index):-
    0 == mod(Index, LengthRow),
    !,
    write(X), nl,
    NewIndex is Index + 1,
    display_board(Xs, LengthRow, NewIndex).
display_board([X|Xs], LengthRow, Index):-
    write(X), write(' '),
    NewIndex is Index + 1,
    display_board(Xs, LengthRow, NewIndex).

```

#### yin\_yang\_restrict.pl

```

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

/*
0 --> black
1 --> white
LastEvaluatorNumber --> last column of penultima line
*/

test_board_4x4([
    A1, A2, A3, A4,
    B1, B2, B3, B4,
    C1, C2, C3, C4,
    D1, D2, D3, D4
]).

test_board_3x3([
    A1, A2, A3,
    B1, B2, B3,
    C1, C2, C3
]).

test_board_2x2([
    A1, A2,
    B1, B2
]).

```

```

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

yin_yang_auto(Board):-
    test_board_4x4(Board),
    length(Board, Length),
    LengthRowAux is sqrt(Length),
    LengthRow is round(LengthRowAux),

    domain(Board, 0, 1),
    setConstrains(Board, 1, Length, LengthRow),
    reset_timer,
    labeling([], Board),
    print_time,
    fd_statistics,

    display_board(Board, LengthRow, 1).

yin_yang_manual(Board, Length):-
    BoardLength is Length * Length,

    length(Board, BoardLength),
    LengthRowAux is sqrt(BoardLength),
    LengthRow is round(LengthRowAux),

    domain(Board, 0, 1),
    setConstrains(Board, 1, BoardLength, LengthRow, [], []),

    reset_timer,
    labeling([], Board),
    print_time,
    fd_statistics,

    display_board(Board, LengthRow, 1).

setEulerPathConstraints(Degrees0, Degrees1):-
    %% apply Euler path to 0s
    count(1, Degrees0, #=, CountDegree1of0),
    count(3, Degrees0, #=, CountDegree3of0),

    count(0, Degrees0, #=, 0),

```

```

CountDegree1of0 + CountDegree3of0 #= 2,

%% apply Euler path to 1s
count(1, Degrees1, #=, CountDegree1of1),
count(3, Degrees1, #=, CountDegree3of1),

count(0, Degrees1, #=, 0),

CountDegree1of1 + CountDegree3of1 #= 2.

% cell that ends predicate. Last cell in the board
setConstrains(Board, Length, Length, LengthRow, Degrees0, Degrees1):-
    % cell that appears in the last row, last column (X = Last, Y =
    Last)
    LeftIndex is Length - 1,
    UpperIndex is Length - LengthRow,

    element(Length, Board, CurrElem),
    element(LeftIndex, Board, LeftElem),
    element(UpperIndex, Board, UpperElem),
    (
        CurrElem #= LeftElem #<=>B,
        CurrElem #= UpperElem #<=>C,
        N #= B+C
    ),

    ((CurrElem #= 0, setEulerPathConstraints([N|Degrees0], Degrees1));
    (CurrElem #= 1, setEulerPathConstraints(Degrees0, [N|Degrees1]))).

setConstrains(Board, 1, Length, LengthRow, Degrees0, Degrees1):-
    % cell that appears in the first row, first column (X = 0, Y =
    0)
    NewIndex is 1 + 1,

    RightIndex is 1 + 1,
    BelowIndex is 1 + LengthRow,
    SEIndex is BelowIndex + 1,

    element(1, Board, CurrElem),
    element(RightIndex, Board, RightElem),
    element(BelowIndex, Board, BelowElem),
    element(SEIndex, Board, SEElem),

    (
        CurrElem #= RightElem #<=>B,
        CurrElem #= BelowElem #<=>C,

```

```

        N #= B+C
    ),

    % special case for 2x2 boards
    nvalue(2, [CurrElem, RightElem, BelowElem, SEElem]),

    ((CurrElem #= 0, setConstrains(Board, NewIndex, Length, LengthRow,
[N|Degrees0], Degrees1));
    (CurrElem #= 1, setConstrains(Board, NewIndex, Length, LengthRow,
Degrees0, [N|Degrees1]))).

setConstrains(Board, LengthRow, Length, LengthRow, Degrees0, Degrees1):-
    % cell that appears in the first row, last column (X = Last, Y
= 0)
    NewIndex is LengthRow + 1,

    LeftIndex is LengthRow - 1,
    BelowIndex is LengthRow + LengthRow,

    element(LengthRow, Board, CurrElem),
    element(LeftIndex, Board, LeftElem),
    element(BelowIndex, Board, BelowElem),

    (
        CurrElem #= LeftElem #<=>B,
        CurrElem #= BelowElem #<=>C,
        N #= B+C
    ),

    ((CurrElem #= 0, setConstrains(Board, NewIndex, Length, LengthRow,
[N|Degrees0], Degrees1));
    (CurrElem #= 1, setConstrains(Board, NewIndex, Length, LengthRow,
Degrees0, [N|Degrees1]))).

setConstrains(Board, Index, Length, LengthRow, Degrees0, Degrees1):-
    % cell that appears in the last row, first column (X = 0, Y = Last)
    LastRowFirstColumn is Length - LengthRow + 1,
    Index == LastRowFirstColumn,
    NewIndex is Index + 1,

    UpperIndex is Index - LengthRow,
    RightIndex is Index + 1,

    element(Index, Board, CurrElem),
    element(UpperIndex, Board, UpperElem),
    element(RightIndex, Board, RightElem),

```

```

(
    CurrElem #= UpperElem #<=>B,
    CurrElem #= RightElem #<=>C,
    N #= B+C
),

((CurrElem #= 0, setConstrains(Board, NewIndex, Length, LengthRow,
[N|Degrees0], Degrees1));
 (CurrElem #= 1, setConstrains(Board, NewIndex, Length, LengthRow,
Degrees0, [N|Degrees1]))).

setConstrains(Board, Index, Length, LengthRow, Degrees0, Degrees1):-
    % cells between first and last line in the last column (excluded)
(X = Last, Y = [1, Last - 1]) (Right side)
    0 =:= mod(Index, LengthRow),
    NewIndex is Index + 1,

    UpperIndex is Index - LengthRow,
    BelowIndex is Index + LengthRow,
    LeftIndex is Index - 1,

    element(Index, Board, CurrElem),
    element(UpperIndex, Board, UpperElem),
    element(BelowIndex, Board, BelowElem),
    element(LeftIndex, Board, LeftElem),

(
    CurrElem #= UpperElem #<=> B,
    CurrElem #= BelowElem #<=> C,
    CurrElem #= LeftElem #<=> D,
    N #= B + C + D
),

((CurrElem #= 0, setConstrains(Board, NewIndex, Length, LengthRow,
[N|Degrees0], Degrees1));
 (CurrElem #= 1, setConstrains(Board, NewIndex, Length, LengthRow,
Degrees0, [N|Degrees1]))).

setConstrains(Board, Index, Length, LengthRow, Degrees0, Degrees1):-
    % cells in the last row between first and last columns (excluded)
(X = [1, Last - 1], Y = Last) (Below Side)
    LastRowIndex is Length - LengthRow,
    Index > LastRowIndex,
    NewIndex is Index + 1,

```

```

LeftIndex is Index - 1,
RightIndex is Index + 1,
UpperIndex is Index - LengthRow,

element(Index, Board, CurrElem),
element(LeftIndex, Board, LeftElem),
element(RightIndex, Board, RightElem),
element(UpperIndex, Board, UpperElem),

(
    CurrElem #= LeftElem #<=> B,
    CurrElem #= RightElem #<=> C,
    CurrElem #= UpperElem #<=> D,
    N #= B + C + D
),

((CurrElem #= 0, setConstrains(Board, NewIndex, Length, LengthRow,
[N|Degrees0], Degrees1));
  (CurrElem #= 1, setConstrains(Board, NewIndex, Length, LengthRow,
Degrees0, [N|Degrees1]))).

setConstrains(Board, Index, Length, LengthRow, Degrees0, Degrees1):-
    % cells in the first column between first and last rows (excluded)
(X = 0, Y = [1, Last - 1]) (Left Side)
    CheckIndex is Index - 1,
    0 =:= mod(CheckIndex, LengthRow),
    NewIndex is Index + 1,

    RightIndex is Index + 1,
    UpperIndex is Index - LengthRow,
    BelowIndex is Index + LengthRow,

    element(Index, Board, CurrElem),
    element(RightIndex, Board, RightElem),
    element(UpperIndex, Board, UpperElem),
    element(BelowIndex, Board, BelowElem),

    (
        CurrElem #= UpperElem #<=> B,
        CurrElem #= RightElem #<=> C,
        CurrElem #= BelowElem #<=> D,
        N #= B + C + D
    ),

    ((CurrElem #= 0, setConstrains(Board, NewIndex, Length, LengthRow,
[N|Degrees0], Degrees1));

```



```

(CurrElem #= 1, setConstrains(Board, NewIndex, Length, LengthRow,
Degrees0, [N|Degrees1]))).

setConstrains(Board, Index, Length, LengthRow, Degrees0, Degrees1):-
    % cells in the first line between first and last columns (excluded)
    (X = [1, Last - 1], Y = 0) (Upper Side)
    Index < LengthRow,
    NewIndex is Index + 1,

    LeftIndex is Index - 1,
    BelowIndex is Index + LengthRow,
    RightIndex is Index + 1,

    element(Index, Board, CurrElem),
    element(LeftIndex, Board, LeftElem),
    element(BelowIndex, Board, BelowElem),
    element(RightIndex, Board, RightElem),

    (
        CurrElem #= LeftElem #<=> B,
        CurrElem #= BelowElem #<=> C,
        CurrElem #= RightElem #<=> D,
        N #= B + C + D
    ),

    ((CurrElem #= 0, setConstrains(Board, NewIndex, Length, LengthRow,
[N|Degrees0], Degrees1));
    (CurrElem #= 1, setConstrains(Board, NewIndex, Length, LengthRow,
Degrees0, [N|Degrees1]))).

setConstrains(Board, Index, Length, LengthRow, Degrees0, Degrees1):-
    NewIndex is Index + 1,

    UpperIndex is Index - LengthRow,
    RightIndex is Index + 1,
    BelowIndex is Index + LengthRow,
    LeftIndex is Index - 1,

    NWIndex is UpperIndex - 1,
    NEIndex is UpperIndex + 1,
    SWIndex is BelowIndex - 1,
    SEIndex is BelowIndex + 1,

    element(Index, Board, CurrElem),
    element(UpperIndex, Board, UpperElem),
    element(RightIndex, Board, RightElem),

```

```

element(BelowIndex, Board, BelowElem),
element(LeftIndex, Board, LeftElem),

element(NWIndex, Board, NWElem),
element(NEIndex, Board, NEElem),
element(SWIndex, Board, SWElem),
element(SEIndex, Board, SEElem),

(
    CurrElem #= UpperElem #<=> B,
    CurrElem #= RightElem #<=> C,
    CurrElem #= BelowElem #<=> D,
    CurrElem #= LeftElem #<=> E,
    N #= B + C + D + E
),

% top left
nvalue(2, [CurrElem, UpperElem, NWElem, LeftElem]),

% top right
nvalue(2, [CurrElem, UpperElem, NEElem, RightElem]),

% bottom left
nvalue(2, [CurrElem, BelowElem, SWElem, LeftElem]),

%bottom right
nvalue(2, [CurrElem, BelowElem, SEElem, RightElem]),

((CurrElem #= 0, setConstrains(Board, NewIndex, Length, LengthRow,
[N|Degrees0], Degrees1));
(CurrElem #= 1, setConstrains(Board, NewIndex, Length, LengthRow,
Degrees0, [N|Degrees1]))).

display_board([], _, _).
display_board([X|Xs], LengthRow, Index):-
    0 == mod(Index, LengthRow),
    !,
    write(X), nl,
    NewIndex is Index + 1,
    display_board(Xs, LengthRow, NewIndex).
display_board([X|Xs], LengthRow, Index):-
    write(X), write(' '),
    NewIndex is Index + 1,
    display_board(Xs, LengthRow, NewIndex).

```