Akari/Light-Up Resolvido Usando Restrições em Programação Lógica

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

FEUP-PLOG, Turma 3MIEIC1, Grupo Akari_5
João Pedro Milano da Silva Cardoso - 200900579
Diogo Alexandre Soares Gomes - 201106586

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, s/n, 4200-465 Porto, Portugal

Dezembro de 2014

Resumo.

O objetivo deste trabalho foi encontrar a solução para um problema de decisão combinatória na forma de um puzzle 2D utilizando a linguagem Prolog. Para isso foram utilizadas restrições para aplicar as regras do jogo Akari/Light-Up ao programa.

Akari consiste num puzzle em que o jogador preenche um tabuleiro quadrado de comprimento variável com luzes de modo a que todos os espaços brancos acabem acessos.

O programa em questão resolve puzzles deste tipo, apresentando a solução final de uma forma visual e compreensiva.

Foi também desenvolvido um gerador de puzzles simples que só contêm espaços pretos.

O trabalho foi realizado com sucesso, sendo que os resultados comprovam a resolução correcta de puzzles de diferentes tamanhos.

Conteúdo

1 Introdução

O presente artigo foi desenvolvido na âmbito da unidade curricular de Programação Lógica do Curso de Mestrado Integrado em Engenharia Informática e de Computação.

O trabalho consiste na elaboração de um programa capaz de resolver tabuleiros do jogo Akari, na linguagem declarativa Prolog, com o objectivo de aprofundar o nosso conhecimento nessa linguagem, mais especificamente em relação a programação lógica usando restrições.

Além do programa para resolver puzzles foi também implementado um simples gerador capaz de criar puzzles de vários tamanhos, os quais apenas contêm espaços pretos.

O artigo está divido nas seguintes principais secções:

Descrição do Problema, onde é descrito em mais detalhe o funcionamento do jogo; Abordagem, onde é descrita a modelação do problema como um problema de restrições; Visualização da Solução, onde é explicada a visualização da solução em modo de texto e Resultados, com resultados que permitem concluir o correcto funcionamento do programa.

2 Descrição do problema

O jogo Akari toma lugar num tabuleiro quadrado de comprimento variável.

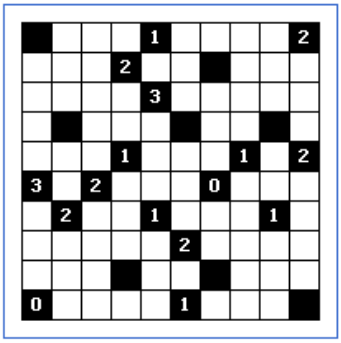


Fig. 1. - Exemplo de Tabuleiro 10x10

Nesse tabuleiro existem três tipos de espaços – vazios, pretos e numerados.

O objectivo do jogo é colocar luzes nos espaços brancos de modo a que todos os espaços brancos sejam iluminados respeitando as seguintes regras:

- Luzes apenas podem ser colocadas nos espaços brancos.
- Uma luz ilumina todos os espaços brancos nas quatro direcções cardinais até encontrar o fim do tabuleiro, um espaço preto ou um espaço numerado.

– Uma luz não pode iluminar outra luz.
– O número de luzes nos espaços adjacentes a um espaço numerado tem de ser igual ao número no espaço.

3 Abordagem

Variáveis de decisão

O predicado de resolução - akari(Rows, Size) - recebe como argumentos uma lista em que cada elemento é uma lista, representando assim o tabuleiro.

As variáveis de decisão são os espaços em branco do tabuleiro inicial. A esses espaços serão atribuídos um de dois valores – 0 se for um espaço em branco e 1 se for uma luz. No entanto, devido à existência dos restantes espaços, o domínio da variável que contém todos os espaços do tabuleiro, Board, é entre 0 e 15. Um espaço preto recebe o número 15 e os espaços numerados de 0 a 4 recebem os números 10 a 14, respectivamente.

O uso dos números 0 para espaços brancos e 1 para luzes facilita a aplicação das restrições, as quais podem ser aplicadas recorrendo ao predicado sum.

Restrições

De modo a cumprir todas as regras foram estabelecidas 5 restrições, considerando que o espaço em branco será a variável Elem e as colunas, linhas, vizinhanças e espaços adjacentes a um bloco numerado listas de variáveis:

- Cada espaço pode conter no máximo uma luz. – sum([Elem], #=<, 1)
- A coluna a que o espaço pertence pode conter no máximo uma luz, de modo a que uma luz não ilumine outra. – sum(Coluna, #<=, 1)
- A linha a que o espaço pertence pode conter no máximo uma luz, de modo a que uma luz não ilumine outra. – sum(Linha, #<=, 1)
- De modo a que o espaço seja iluminado, a vizinhança do espaço – a junção da sua coluna com a sua linha, obtendo todos os espaços visíveis a partir do mesmo – tem de conter pelo menos uma luz. – sum(Viz, #>=, 1)
- Para os espaços numerados, a soma total dos quatro espaços adjacentes tem de ser igual ao número no espaço. – sum(Adj, #=, Num)

A vizinhança é delimitada por espaços pretos ou numerados e os limites do tabuleiro. A aplicação destas cinco restrições são suficientes para resolver um tabuleiro de akari, de tamanho variável.

Estratégia de pesquisa

A estratégia de labeling utilizada é a estratégia MIN. As variáveis são selecionadas começando pela variável mais à esquerda possível com o valor de limite mais baixo. Testes efectuados depois da realização do trabalho, com os resultados apresentados na secção 5 permite verificar qual a melhor estratégia.

4 Vizualização

Um tabuleiro de akari resolvido é impresso no seguinte formato:

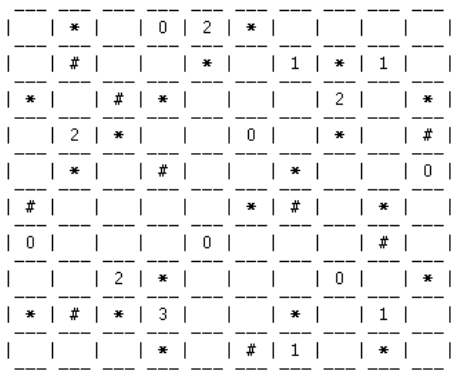


Fig. 2. – Exemplo de Tabuleiro Resolvido

Devido á natureza do problema a resolver, o predicado de impressão, printBoard(Board, Size), suporta a impressão de listas de comprimentos diferentes.

Os elemento presentes a impressão têm o seguinte significado:

- Espaço em branco – Espaço vazio, iluminado na solução
- Asterisco (*) – Luz
- Cardinal (#) – Espaço preto
- Números de 0 a 4 – Espaços numerados

5 Resultados

Depois da realização do trabalho foram executados testes na tentativa de determinar a melhor estratégia a aplicar ao predicado labeling de modo a tornar a resolução de problamas o mais rápida possível.

Foram executados 10 testes para cada estratégia, com um tabuleiro aleatoriamente gerado de 40x40 espaços, sendo a média de cada estratégia apresentada no seguinte gráfico. Ao comparar os dados em relação ao tabuleiro de 25x25 presente no código fonte, a pior estratégia foi FFC, as restantes produziram resultados semelhantes.

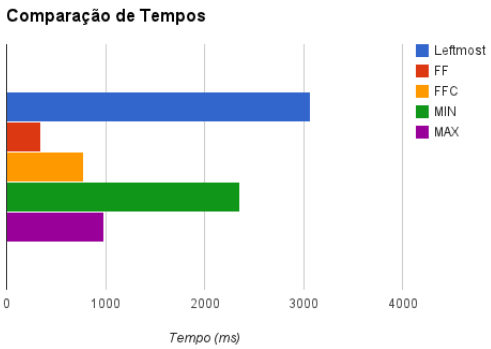


Fig. 3. Comparação de tempos de estratégias diferentes

Também se fizeram alguns testes em relação ao tempo que o programa encontra uma solução em relação ao tamanho do tabuleiro. Para este teste foi utilizado a parte de geração de puzzles para diferentes tamanhos em que para cada tamanho se efetuou 15 testes. O tempo começa por estar á volta dos 0ms até aos tabuleiros de tamanho 35x35 mas a partir daí percebe-se que o tempo aumenta de forma quase expoxencial.

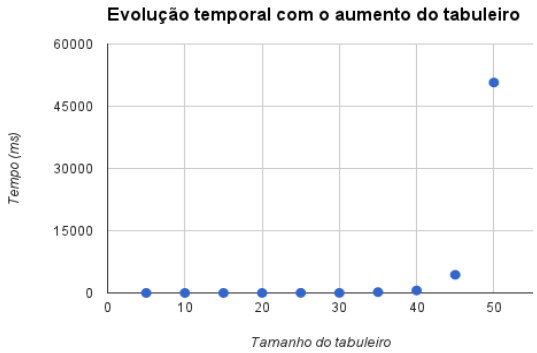


Fig. 4. Média do tempo de resolução relativamente ao tamanho do tabuleiro

Utilizando um site[1], é possível gerar tabuleiros de 3 tamanhos diferentes (7x7, 10x10 e 25x25) e três niveis de dificuldade. Estes puzzles têm uma só solução. É apresentado a seguir um tabuleiro de 7x7, dificuldade difícil, e a solução apresentada pelo programa.

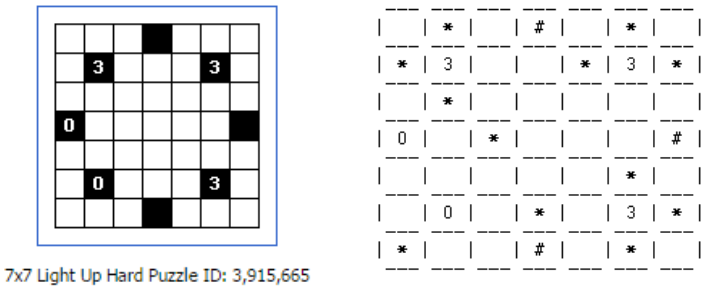


Fig. 5. Tabuleiro 7x7 e Solução do Programa

O próprio site serve também para uma pessoa poder resolver o puzzle manualmente. Ao colocar no site a solução como apresentada pelo programa produz o seguinte resultado, podendo assim ser verificado o correcto funcionamento do programa:

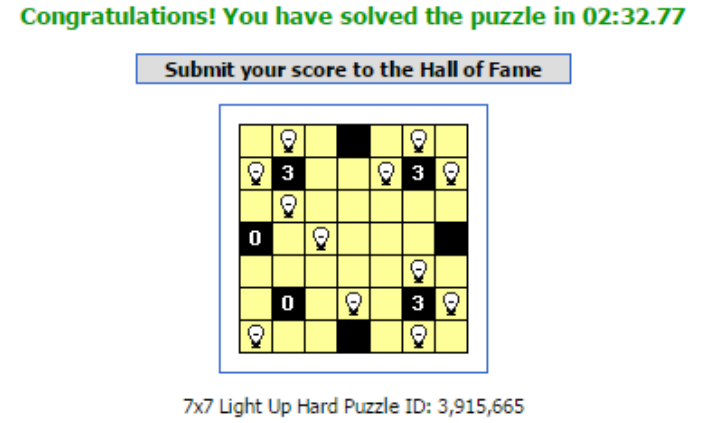
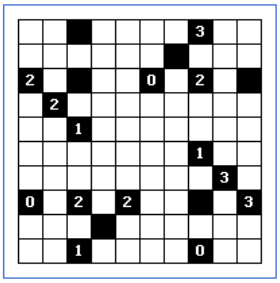


Fig. 6. Tabuleiro 7x7 Resolvido

Testando agora com um tabuleiro de 10x10 difícil.



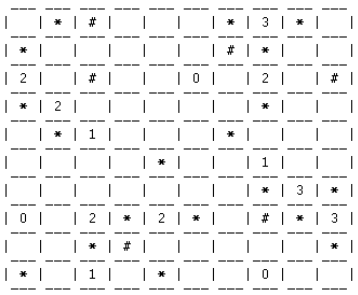


Fig. 7. Tabuleiro de 10x10 e solução do programa

Fig. 8. Tabuleiro 10x10 Resolvido

6 Conclusão

É possível concluir que o projecto foi completado com sucesso, sendo capaz de resolver correctamente puzzles aplicando restrições lógicas de Prolog.

Embora os tempos de resolução sejam mínimos, na ordem os milisegundos, existem ainda melhorias a serem executadas do ponto de vista de performance, como por exemplo a utilização do predicado transpose, o que permitiria menos código para aplicar as restrições. Um outro aspecto que poderia melhorar a performance seria uma estratégia de pesquisa personalizada para aplicar ao labeling.

7 Bibliografia

1. Light-Up Online http://www.puzzle-light-up.com/
2. Documentação SICStus https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/

8 Anexos

Código

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% MODULES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:-use_module(library(clpfd)).
:-use_module(library(lists)).
:-use_module(library(random)).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% LEGEND %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% _   -> Empty space <- 0
% 1   -> Light <- *
% 15  -> Black Block <- #
% 10 -> Zero Block <- 0
% 11 -> One Block <- 1
% 12 -> Two Block <- 2
% 13 -> Three Block <- 3
% 14 -> Four Block <- 4
% leftmost, min, max, ff, ffc
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% INITIALIZATION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

start:-
    akari([
    [_,_,_,10,_,_,_],
    [_,13,_,_,_,15,_],
    [_,_,15,_,15,_,_],
    [15,_,_,15,_,_,13],
    [_,_,12,_,15,_,_],
    [_,12,_,_,_,15,_],
    [_,_,_,11,_,_,_]],7).

start2:-
    akari([
    [_,_,_,10,12,_,_,_,_,_],
    [_,15,_,_,_,_,11,_,11,_],
```

```
    [_,_,15,_,_,_,_,12,_,_],
    [_,12,_,_,_,10,_,_,_,15],
    [_,_,_,15,_,_,_,_,_,10],
    [15,_,_,_,_,_,15,_,_,_],
    [10,_,_,_,10,_,_,_,15,_],
    [_,_,12,_,_,_,_,10,_,_],
    [_,15,_,13,_,_,_,_,11,_],
    [_,_,_,_,_,15,11,_,_,_]],10).

start3:-
    akari([
      [_,_,_,_,_,11,_,_,_,_,_,15,_,_,_,_,_,_,_,11,_,_,11,_],
      [15,_,_,15,_,15,_,11,_,15,_,12,_,_,12,_,15,_,12,_,_,_,_,_,_],
      [_,_,11,_,_,_,11,15,_,_,_,_,_,_,_,_,_,15,15,_,_,15,12,_,_],
      [_,_,12,_,_,_,_,_,_,10,15,_,_,10,_,11,_,_,_,15,_,_,_,15,_],
      [15,_,_,_,_,_,_,_,_,_,_,_,_,_,_,15,_,_,_,_,_,_,_,_,_],
      [_,_,_,15,_,11,_,15,_,_,15,_,_,_,_,10,15,_,13,15,_,_,_,15,15],
      [_,11,15,_,_,15,_,_,_,15,11,_,15,_,_,_,_,_,_,_,_,_,15,_,_],
      [_,_,11,_,_,_,_,12,15,_,15,_,_,15,_,13,_,15,_,15,_,_,10,15,_],
      [_,11,_,_,15,15,_,_,_,12,_,15,11,_,15,_,_,15,_,_,_,_,_,_,_],
      [_,_,_,10,_,15,_,13,_,_,_,_,15,_,_,_,15,_,15,_,_,15,_,11,_],
      [_,11,_,_,_,_,_,_,15,_,_,_,15,12,_,_,_,15,15,10,_,10,_,_,_],
      [_,_,_,11,_,_,_,11,_,_,15,_,15,_,_,_,11,_,_,_,_,_,_,10,15],
      [_,_,_,_,_,_,_,15,_,15,15,15,12,15,15,15,15,11,_,12,_,_,_,_,_,_],
      [15,15,_,_,_,_,_,_,15,_,_,_,15,_,15,_,_,12,_,_,_,15,_,_,_],
      [_,_,_,10,_,15,15,13,_,_,_,15,15,_,_,_,11,_,_,_,_,_,_,15,_],
      [_,15,_,15,_,_,15,_,12,_,_,_,15,_,_,_,_,15,_,10,_,15,_,_,_],
      [_,_,_,_,_,_,_,15,_,_,15,_,15,15,_,15,_,_,_,11,11,_,_,15,_],
      [_,13,11,_,_,15,_,15,_,15,_,13,_,_,11,_,15,15,_,_,_,_,15,_,_],
      [_,_,15,_,_,_,_,_,_,_,_,13,_,15,13,_,_,_,15,_,_,15,12,_],
      [15,11,_,_,_,10,15,_,15,10,_,_,_,_,12,_,_,10,_,15,_,15,_,_,_],
      [_,_,_,_,_,_,_,_,15,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,11],
      [_,15,_,_,_,13,_,_,_,15,_,10,_,_,15,15,_,_,_,_,_,_,11,_,_],
      [_,_,11,10,_,_,15,12,_,_,_,_,_,_,_,_,_,15,11,_,_,_,10,_,_],
      [_,_,_,_,_,_,15,_,15,_,15,_,_,15,_,15,_,12,_,15,_,11,_,_,15],
      [_,15,_,_,10,_,_,_,_,_,_,_,_,11,_,_,_,_,_,15,_,_,_,_,_]],25).

start4:- akari([
        [15,15,15],
        [15,15,15],
        [15,15,15]],3).

start5:-
    akari([
    [_,_,_,15,_,_,_],
```

```
    [_,13,_,_,_,13,_],
    [_,_,_,_,_,_,_],
    [10,_,_,_,_,_,15],
    [_,_,_,_,_,_,_],
    [_,10,_,_,_,13,_],
    [_,_,_,15,_,_,_]],7).

start6:-
    akari([
    [_,_,15,_,_,_,_,13,_,_],
    [_,_,_,_,_,_,_,15,_,_,_],
    [12,_,15,_,_,10,_,12,_,15],
    [_,12,_,_,_,_,_,_,_,_],
    [_,_,11,_,_,_,_,_,_,_],
    [_,_,_,_,_,_,_,11,_,_],
    [_,_,_,_,_,_,_,_,13,_],
    [10,_,12,_,12,_,_,15,_,13],
    [_,_,_,15,_,_,_,_,_,_],
    [_,_,11,_,_,_,_,10,_,_]],10).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%% MAIN %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
akari(Rows, Size) :-
  length(Rows, Size), maplist(length_list(Size), Rows),
  append(Rows, Board),
  domain(Board, 0, 15),
  processRows(Rows, Rows, 0-0, Size),
  reset_timer,
  labeling([], Board),
  print_time,
  fd_statistics,
  printBoard(Board, Size).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%% PROCESSING ROWS -> PLURAL %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
processRows([], _Board, _Row-_Col, _Size).

processRows([H|T], Board, Row-Col, Size):-
  processRow(H, Board, Row-Col, Size, []),
  NewRow is Row + 1,
  processRows(T, Board, NewRow-0, Size).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% PROCESSING ROW -> SINGULAR %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
processRow([], _Board, _Row-_Col, _Size, _VarList).
```

```
processRow([H|T], Board, Row-Col, Size, _VarList):-
  H == 15,
  NewCol is Col + 1,
  processRow(T, Board, Row-NewCol, Size, []).

processRow([H|T], Board, Row-Col, Size, _VarList):-
  compare(H),
  hintRestriction(Board, Row-Col, H, Size),
  NewCol is Col + 1,
  processRow(T, Board, Row-NewCol, Size, []).

processRow([H|T], Board, Row-Col, Size, VarList):-
  insertAtEnd(H, [], L),
  sum(L, #=<, 1), % each space can have at most one light
  gatherLeft(Board, Row-Col, Row-Col, Size, VarList),
  NewCol is Col + 1,
  processRow(T, Board, Row-NewCol, Size, []).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% RESTRICT HINTS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
hintRestriction(Board, Row-Col, NumberHigh, Size):-
  Number is NumberHigh - 10,
  NewRow1 is Row - 1,
  getUpSquare(Board, NewRow1-Col, Size, US),
  append(US, [], List),
  NewRow2 is Row + 1,
  getDownSquare(Board, NewRow2-Col, Size, DS),
  append(DS, List, List2),
  NewCol1 is Col - 1,
  getLeftSquare(Board, Row-NewCol1, Size, LS),
  append(LS, List2, List3),
  NewCol2 is Col + 1,
  getRightSquare(Board, Row-NewCol2, Size, RS),
  append(RS, List3, FinalList),
  sum(FinalList, #=, Number).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% GET UP SQUARE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getUpSquare(_Board, Row-_Col, _Size, US):-
  Row < 0,
  US = [].

getUpSquare(Board, Row-Col, _Size, US):-
```

```
  getElem(Board, Row-Col, Elem),
  compare(Elem),
  US = [].

getUpSquare(Board, Row-Col, _Size, US):-
  getElem(Board, Row-Col, Elem),
  US = [Elem].
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% GET DOWN SQUARE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getDownSquare(_Board, Row-_Col, Size, DS):-
  Row >= Size,
  DS = [].

getDownSquare(Board, Row-Col, _Size, DS):-
  getElem(Board, Row-Col, Elem),
  compare(Elem),
  DS = [].

getDownSquare(Board, Row-Col, _Size, DS):-
  getElem(Board, Row-Col, Elem),
  DS = [Elem].
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% GET LEFT SQUARE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getLeftSquare(_Board, _Row-Col, _Size, LS):-
  Col < 0,
  LS = [].

getLeftSquare(Board, Row-Col, _Size, LS):-
  getElem(Board, Row-Col, Elem),
  compare(Elem),
  LS = [].

getLeftSquare(Board, Row-Col, _Size, LS):-
  getElem(Board, Row-Col, Elem),
  LS = [Elem].
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% GET RIGHT SQUARE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
getRightSquare(_Board, _Row-Col, Size, RS):-
  Col >= Size,
  RS = [].
```

```
getRightSquare(Board, Row-Col, _Size, RS):-
  getElem(Board, Row-Col, Elem),
  compare(Elem),
  RS = [].

getRightSquare(Board, Row-Col, _Size, RS):-
  getElem(Board, Row-Col, Elem),
  RS = [Elem].
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%% GATHER LEFT %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

gatherLeft(Board, ORow-OCol, _CRow-CCol, Size, RowList):-
  CCol < 0,
  NewCol is OCol + 1,
  gatherRight(Board, ORow-OCol, ORow-NewCol, Size, RowL-
ist).

gatherLeft(Board, ORow-OCol, CRow-CCol, Size, RowList):-
  getElem(Board, CRow-CCol, Elem),
  compare(Elem),
  NewCol is OCol + 1,
  gatherRight(Board, ORow-OCol, ORow-NewCol, Size, RowL-
ist).

gatherLeft(Board, ORow-OCol, CRow-CCol, Size, RowList):-
  getElem(Board, CRow-CCol, Elem),
  insertAtEnd(Elem, RowList, NewRowList),
  NewCol is CCol - 1,
  gatherLeft(Board, ORow-OCol, CRow-NewCol, Size,
NewRowList).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% GATHER RIGHT %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

gatherRight(Board, ORow-OCol, _CRow-CCol, Size, RowL-
ist):-
  CCol >= Size,
  sum(RowList, #=<, 1),
  gatherUp(Board, ORow-OCol, ORow-OCol, Size, RowList,
[]).

gatherRight(Board, ORow-OCol, CRow-CCol, Size, RowList):-
  getElem(Board, CRow-CCol, Elem),
```

```
  compare(Elem),
  sum(RowList, #=<, 1),
  gatherUp(Board, ORow-OCol, ORow-OCol, Size, RowList,
[]).

gatherRight(Board, ORow-OCol, CRow-CCol, Size, RowList):-
  getElem(Board, CRow-CCol, Elem),
  insertAtEnd(Elem, RowList, NewRowList),
  NewCol is CCol + 1,
  gatherRight(Board, ORow-OCol, CRow-NewCol, Size,
NewRowList).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% GATHER UP %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

gatherUp(Board, ORow-OCol, CRow-_CCol, Size, VarList,
ColList):-
  CRow < 0,
  NewRow is ORow + 1,
  gatherDown(Board, ORow-OCol, NewRow-OCol, Size,
VarList, ColList).

gatherUp(Board, ORow-OCol, CRow-CCol, Size, VarList, Col-
List):-
  getElem(Board, CRow-CCol, Elem),
  compare(Elem),
  NewRow is ORow + 1,
  gatherDown(Board, ORow-OCol, NewRow-OCol, Size,
VarList, ColList).

gatherUp(Board, ORow-OCol, CRow-CCol, Size, VarList, Col-
List):-
  getElem(Board, CRow-CCol, Elem),
  insertAtEnd(Elem, VarList, NewVarList),
  insertAtEnd(Elem, ColList, NewColList),
  NewRow is CRow - 1,
  gatherUp(Board, ORow-OCol, NewRow-CCol, Size,
NewVarList, NewColList).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% GATHER DOWN %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

gatherDown(_Board, _ORow-_OCol, CRow-_CCol, Size,
VarList, ColList):-
  CRow >= Size,
```

```
  sum(ColList, #=<, 1),
  sum(VarList, #>=, 1).

gatherDown(Board, _ORow-_OCol, CRow-CCol, _Size, VarList,
ColList):-
  getElem(Board, CRow-CCol, Elem),
  compare(Elem),
  sum(ColList, #=<, 1),
  sum(VarList, #>=, 1).


gatherDown(Board, ORow-OCol, CRow-CCol, Size, VarList,
ColList):-
  getElem(Board, CRow-CCol, Elem),
  insertAtEnd(Elem, VarList, NewVarList),
  insertAtEnd(Elem, ColList, NewColList),
  NewRow is CRow + 1,
  gatherDown(Board, ORow-OCol, NewRow-CCol, Size,
NewVarList, NewColList).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%% GENERATOR %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
generate(Size):-
  length(Rows, Size), maplist(length_list(Size), Rows),
  append(Rows, Board),
  domain(Board, 0, 15),
  boardRestrictions(Rows, Size),
  labeling([], Board),
  printBoard(Board, Size), nl,
  copy(Rows, _, Size, NewFinal),
  akari(NewFinal, Size).

boardRestrictions(Rows, Size):-
  runRows(Rows, Rows, 0-0, Size).

runRows([], _Board, _Row-_Col, _Size).

runRows([H|T], Board, Row-Col, Size):-
  runRow(H, Board, Row-Col, Size),
  NewRow is Row + 1,
  runRows(T, Board, NewRow-0, Size).

runRow([], _Board, _Row-_Col, _Size).

runRow([H|T], Board, Row-Col, Size):-
```

```
  random(0, 2, Random),
  processSquare(H, Board, Row-Col, Random, Size),
  NewCol is Col + 1,
  runRow(T, Board, Row-NewCol, Size).

processSquare(Elem, Board, Row-Col, Random, Size):-
  Random == 1,
  random(0, 2, NewRandom),
  processSquare2(Elem, Board, Row-Col, NewRandom, Size).

processSquare(_Elem, _Board, _Row-_Col, _Random, _Size).

processSquare2(Elem, _Board, _Row-_Col, Random, _Size):-
  Random == 0,
  sum([Elem], #=, 15).

processSquare2(Elem, _Board, _Row-_Col, _Random, _Size):-
  sum([Elem], #=, 0).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% PRINT SECTION %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

printLine(0).
printLine(Count):-
  write(' ---'),
  Count2 is Count-1,
  printLine(Count2).

% PRINT ELEMENTS OF THE BOARD
printElem(Piece) :- Piece=0, write('   ').
printElem(Piece) :- Piece=1, write(' * ').
printElem(Piece) :- Piece=15, write(' # ').
printElem(Piece) :- Piece=10, write(' 0 ').
printElem(Piece) :- Piece=11, write(' 1 ').
printElem(Piece) :- Piece=12, write(' 2 ').
printElem(Piece) :- Piece=13, write(' 3 ').
printElem(Piece) :- Piece=14, write(' 4 ').


printBoard(Board, Size) :- nl, printLine(Size), nl,print-
List(Board, Size, Size).

printList([],Size,_) :-
  write('|'), nl,
  printLine(Size), nl.
```

```
printList([H|T],Size, 0) :-
  write('|'), nl,
  printLine(Size), nl,
  printList([H|T],Size,Size).

printList([H|T], Size, Count) :-
  NewCount is Count - 1,
  write('|'),
  printElem(H),
  printList(T, Size, NewCount).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%% UTILITIES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
length_list(L, Ls) :- length(Ls, L).

insertAtEnd(X,[ ],[X]).
insertAtEnd(X,[H|T],[H|Z]) :- insertAtEnd(X,T,Z).

getElem(Board, Row-Col, Piece):-
  nth0(Row, Board, ColList),
  nth0(Col, ColList, Piece).

compare(Elem) :- Elem == 15.
compare(Elem) :- Elem == 10.
compare(Elem) :- Elem == 11.
compare(Elem) :- Elem == 12.
compare(Elem) :- Elem == 13.
compare(Elem) :- Elem == 14.

copy(Initial, Final, Size, NewFinal):-
  length(Final, Size), maplist(length_list(Size), Final),
  length(NewFinal, Size), maplist(length_list(Size), New-
Final),
  copyRows(Initial, Final, 0-0, Size, NewFinal).

copyRows([], _Final, _Row-_Col, _Size, _NewFinal).

copyRows([H|T], Final, Row-Col, Size, NewFinal):-
  copyRow(H, Final, Row-Col, Size, NewFinal),
  NewRow is Row + 1,
  copyRows(T, Final, NewRow-0, Size, NewFinal).

copyRow([], _Final, _Row-_Col, _Size, _NewFinal).
```

```
copyRow([H|T], Final, Row-Col, Size, NewFinal):-
  compare(H),
  changeBoard(Final, Row-Col, H, NewFinal),
  NewCol is Col + 1,
  copyRow(T, NewFinal, Row-NewCol, Size, NewFinal).

copyRow([_H|T], Final, Row-Col, Size, NewFinal):-
  NewCol is Col + 1,
  copyRow(T, Final, Row-NewCol, Size, NewFinal).

changeBoard(Board, Row-Col, Piece, NewBoard) :-
  changeBoard2(Board, 0, Col, Piece, Row, NewBoard),!.

changeBoard2([Head|Tail], FinalRow, Col , Piece, Fi-
nalRow, [NewHead|Tail]):-
  replace(Head, Col, Piece, NewHead).

changeBoard2([Head|Tail], Row, Col , Piece, FinalRow,
[Head|NewTail]) :-
  Row \= FinalRow,
  NewRow is Row+1,
  changeBoard2(Tail, NewRow, Col, Piece, FinalRow, New-
Tail).


replace([_|Tail], 0, Element, [Element|Tail]).

replace([Head|Tail], Col, Element, [Head|Rest]):-
  Col > 0, NewCol is Col-1,
  replace(Tail, NewCol, Element, Rest), !.

replace(L, _, _, L).
```

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