Resolução de um Problema de Decisão usando Lógica com Restrições - Doors

André Esteves e Luís Silva

FEUP-PLOG
Turma 3MIEIC1
Grupo Doors_2

Resumo. Este trabalho consiste no desenvolvimento de uma aplicação que crie e resolva puzzles do tipo *Doors*, de tamanho variável, definido pelo utilizador. Esta aplicação foi desenvolvida na linguagem Prolog, fazendo uso de lógica com restrições. A aplicação desenvolvida cumpre, na sua generalidade, os objetivos propostos.

1 Introdução

Os objetivos deste trabalho passavam pelo desenvolvimento de uma aplicação que resolvesse problemas do tipo *Doors*, passados pelo utilizador num formato predeterminado. Além disso, a solução deveria ser visualizada em modo de texto, de forma a que seja fácil a verificação da validade da solução. Adicionalmente, era valorizada a criação dinâmica de problemas, isto é, a geração aleatória do puzzle a resolver.

Um puzzle do tipo *Doors* é um puzzle cujo objetivo consiste em perceber que portas estão abertas ou fechadas numa grelha de divisões.

2 Descrição do Problema

Um puzzle do tipo *Doors* consiste numa tabela quadrada preenchida por números. Cada célula da tabela representa uma sala. Entre cada duas salas adjacentes ortogonalmente existe uma porta, que pode estar aberta ou fechada (representada, respetivamente, por um espaço vazio ou uma linha). O número dentro de cada sala representa o número de salas visíveis através de portas abertas nas quatro direções ortogonais, incluindo a própria sala (cada sala pode, no mínimo, "ver-se" a si mesma).

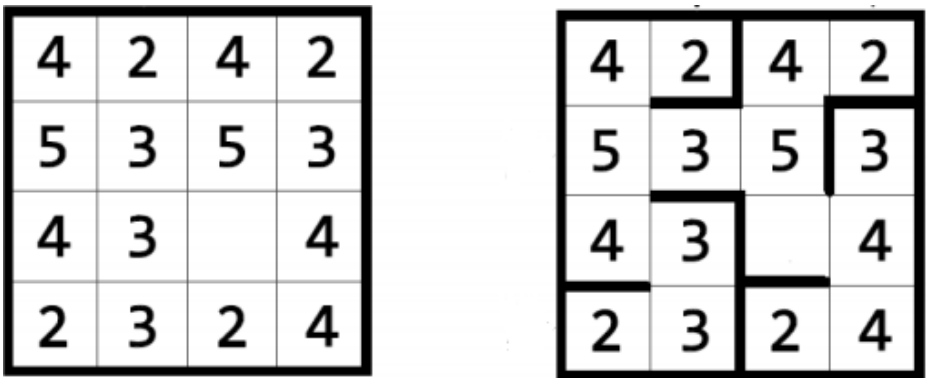


Fig. 1. Puzzle do tipo *Doors* por resolver (à esquerda) e resolvido (à direita)

Um puzzle do tipo Doors por resolver apresenta todas as portas como abertas. O objetivo passa por desenhar as portas que estão fechadas. Como é possível ver no exemplo resolvido, o quarto no canto superior esquerdo consegue ver-se a si mesmo, um quarto à direita e dois abaixo, totalizado quatro.

3 Abordagem

Internamente, um puzzle do tipo *Doors* é representado por três matrizes: uma matriz N por N, onde N é o tamanho do puzzle, que representa os quartos e contém nela os valores correspondentes a cada quarto, e duas matrizes N por N – 1, que representam, respetivamente, as portas horizontais e verticais, sendo que a cada porta corresponde o valor 0 (porta aberta) ou o valor 1 (porta fechada).

3.1 Variáveis de Decisão

As variáveis de decisão para a resolução do problema são os valores pertencentes às duas matrizes que contém os valores associados às portas horizontais e verticais. Como foi dito anteriormente, estas variáveis podem tomar como valor 0 ou 1.

3.2 Restrições

Ambas as matrizes contendo os valores associados às portas têm que ter dimensões N por N – 1, onde N é o tamanho da matriz contendo os valores correspondentes aos quartos, definida previamente. Estas restrições são implementadas em *SICStus Prolog* fazendo uso do predicado *length*. Para além disso, cada variável contida nas matrizes das portas tem que ter valor 0 ou 1. Esta restrição é implementada em *SICStus Prolog* fazendo uso do predicado *domain*. As restantes restrições são dependentes dos valores presentes na matriz dos quartos. Cada quatro influencia as portas presentes na sua linha e na sua coluna da seguinte forma:

Valor do quarto = 1 + Quartos visíveis nas quatro direções

O valor de quartos visíveis em cada direção é calculado recursivamente da seguinte forma:

- Se é o último quarto da linha/coluna, o valor de quartos visíveis é 0
- Se não for o último quarto da linha, então:

Porta = 1 ^ Valor = 0 (a porta está fechada)

ou

Porta = 0 ^ Valor = 1 + Valor do próximo quarto (a porta está aberta)

Esta restrição é implementada em *SICStus Prolog* fazendo uso de predicados recursivos e dos operadores #=, #\/ e #/\.

3.3 Função de Avaliação

Não existe nenhuma função que avalie diretamente se o resultado obtido é válido. No entanto, caso o *puzzle* resolvido tenha sido criado dinamicamente pela aplicação e não introduzido pelo utilizador, é possível fazer uma verificação com os predicados existentes. O predicado *createPuzzle*, que cria um puzzle por resolver, devolve a sua solução nas variáveis *Lines* e *Columns*.

```
createPuzzle(Size, Puzzle, Lines, Columns):-
    createGrid(Lines,Columns, Size),
    createEmptyPuzzle(Size,EmptyPuzzle),
    fillPuzzle(EmptyPuzzle, Lines, Columns, 0, 0, Size,
Puzzle).
```

Estes valores não são utilizados pela aplicação quando esta faz uso deste predicado, mas foram mantidos como valores de retorno para existir a possibilidade de estes serem comparados com as matrizes de portas geradas pela aplicação durante a resolução. Esta funcionalidade não faz, no entanto, parte da implementação atual da aplicação.

3.4 Estratégia de Pesquisa

No sentido de testar qual a melhor estratégia de pesquisa, testamos a aplicação com uma série de *puzzles* predefinidos, analisando o número de *backtrackings* realizados na tentativa de o resolver.

Os puzzles usados foram os seguintes:

4x4	_____ |7 4 4 4| |4 1 2 2| |4 3 4 4| |5 2 2 2|

5x5	_____ | 2 6 1 3 3 | | 3 7 3 3 3 | | 3 5 1 4 4 | | 4 6 2 3 2 | | 3 8 4 5 5 |
6x6	_____ | 3 3 2 6 4 4 | | 3,3,2,5,3,2 | | 4,2,5,5,2,2 | | 4,3,6,6,4,3 | | 6,5,7,5,2,3 | | 1,1,5,3,2,4 |
7x7	_____ | 5 6 5 3 2 2 6 | | 4 5 5 4 3 1 5 | | 3 4 5 4 4 1 5 | | 3 5 1 2 2 2 6 | | 3 3 3 3 5 3 6 | | 3 3 3 3 5 4 4 | | 2 3 4 4 5 3 4 |

Os resultados obtidos estão presentes na seguinte tabela:

Op. labeling / **Tamanho**	**[sem opção]**	**ff**	**ffc**	**bisect**	**step**
4x4	48	48	48	48	48
5x5	96	96	96	96	96
6x6	3864	3864	3864	3864	3864
7x7	33624	33624	33624	33624	33624

Como é possível ver pelos resultados, nenhuma opção de pesquisa aumentou a eficiência dos resultados. Devido a este facto, optámos por deixar o *labeling* sem opção.

4 Visualização da Solução

```
drawPuzzle(Puzzle, Lines, _, X, X) :- !,
    nth1(X, Puzzle, Number),
    nth1(X, Lines, Line),
    write('|'),
    drawNumber(Number, Line, X, 1).
```

```
drawPuzzle(Puzzle, Lines, Columns, Size, X) :-
    nth1(X, Puzzle, Number),
    nth1(X, Lines, Line),
    write('|'),
    drawNumber(Number, Line, Size, 1),
    getLine(Columns, X, Column),
    write('|'),
    drawLine(Column),
    NextX is X + 1,
    drawPuzzle(Puzzle, Lines, Columns, Size, NextX).
drawPuzzle(Puzzle, Lines, Columns, Size) :-
    drawLimits(Size),
    drawPuzzle(Puzzle, Lines, Columns, Size, 1),
    drawLimits(Size),
    nl, nl.
```

O predicado *drawPuzzle* é responsável por desenhar o puzzle *Puzzle* de tamanho *Size*, cuja solução está contida nas matrizes *Lines* e *Columns*. Caso o objetivo seja mostrar o puzzle não-resolvido, devem ser passados como argumentos em *Lines* e *Columns* matrizes de tamanho apropriado contendo apenas zeros (ou seja, portas abertas). Estas matrizes "vazias" podem ser geradas usando o predicado *createEmptyGrid*.

O predicado *drawPuzzle* percorre *Puzzle*, desenhado alternadamente, linhas contendo os valores dos quartos alternados com as portas horizontais, e linhas contendo as portas verticais.

```
 ---------------------
| 3   1   1   5   6   3 |
|                       |
| 3   4   3   6   7   5 |
|                       |
| 4   4   2   6   7   5 |
|                       |
| 3   4   4   4   5   3 |
|                       |
| 3   1   3   2   6   4 |
|                       |
| 3   1   3   2   2   4 |
 ---------------------
```

Fig. 2. Puzzle por resolver representado pelo predicado *drawPuzzle*

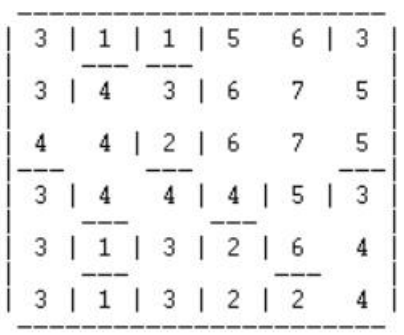
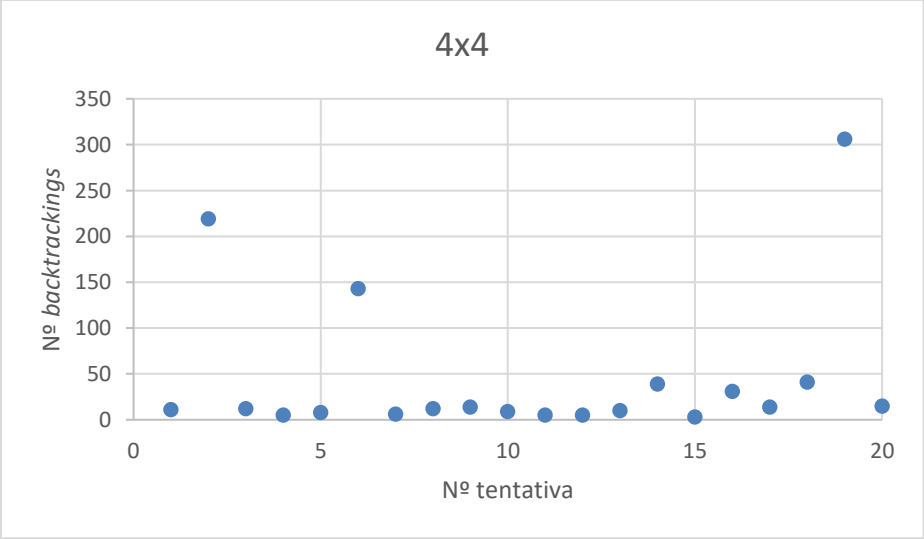


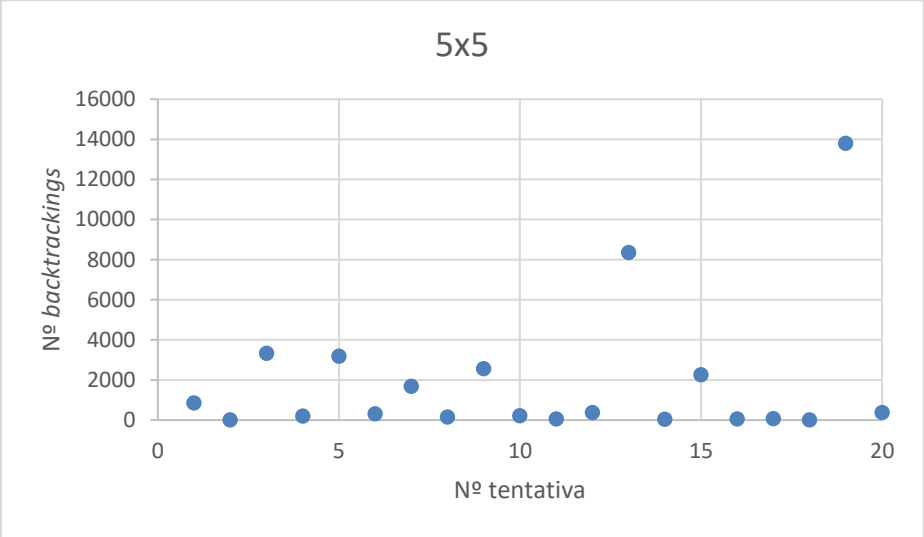
Fig. 3. Puzzle da Fig.2 resolvido representado pelo predicado *drawPuzzle*

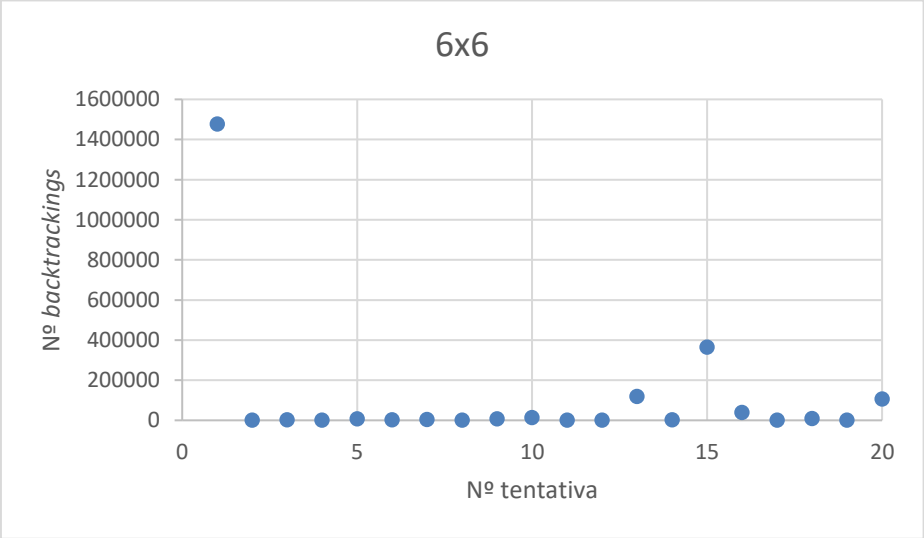
5 Resultados

É possível visualizar no Anexo 8.2 exemplos de execução da aplicação para diferentes tamanhos de *puzzle*. Para *puzzles* de tamanho superior a 8, não existem exemplos de execuções bem-sucedidas devido ao facto de que a aplicação já não os consegue resolver num período de tempo aceitável.

No sentido de analisar a eficiência do código, a aplicação foi executada várias vezes e para diferentes *puzzles* de tamanho diferente. Seguem gráficos que apresentam o número de *backtrackings* executados pela aplicação para *puzzles* de cada tamanho (os *puzzles*, apesar de terem o mesmo tamanho, não são iguais):







Como é possível ver, existe um elevado espectro de número de *backtracking* independentemente do tamanho do *puzzle*.

6 Conclusões e Trabalho Futuro

Podemos concluir que o trabalho foi concluído com sucesso em todos os aspetos propostos. Possíveis melhoramentos futuros poderiam incluir aumentar o nível de eficiência, permitindo resolver *puzzles* de tamanho superior, assim como reduzir a elevada variabilidade de valores de eficiência para *puzzles* do mesmo tamanho.

7 Referencias

1. Constraint Logic Programming over Finite Domains, version 3.7.1, https://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_33.html, consultado pela última vez em 2018/12/20.
2. Constraint Logic Programming over Finite Domains, version 4.1.0, https://sicstus.sics.se/sicstus/docs/4.1.0/html/sicstus/lib_002dclpfd.html, consultado pela última vez em 2018/12/20.

8 Anexos

8.1 Código Fonte

doors.pl

```
:- consult('display.pl').
:- consult('utility.pl').
:- consult('create.pl').
:- consult('solve.pl').

doors(N) :-
    prompt(_, ''),
    % creates and displays a puzzle with N lines and col-
umns
    nl, write('Puzzle:'), nl, nl,
    createPuzzle(N, Puzzle, _, _),
    createEmptyGrid(EmptyLines, EmptyCols, N),
    drawPuzzle(Puzzle, EmptyLines, EmptyCols, N),
    write('Press ENTER to show solution.'),
    % reads ENTER from input
    get_code(C),
    if_then_else(C =:= 10, true, skip_line),
    % solves puzzle
    solvePuzzle(Puzzle, Sol, N),
    % parses solution
    length(Sol, L),
    Length is L div 2,
    length(AuxLines, Length),
    length(AuxCols, Length),
    append(AuxLines, AuxCols, Sol),
    AuxLength is N - 1,
    listToMatrix(AuxLines, Lines, N, AuxLength),
    listToMatrix(AuxCols, Cols, N, AuxLength),
    % displays solution
```

```
    nl, write('Solution:'), nl, nl,
    drawPuzzle(Puzzle, Lines, Cols, N).
```

create.pl

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

/*
    For each line of Sol (N lines), it maps it with val-
ues 1 or 0.
*/
createSolution(Sol, N) :-
    NewN is N - 1,
    length(Sol, N),
    maplist(createLine(NewN), Sol).

/*
    Fills a certain line of the solution with 1's (closed
doors) and 0's (open doors)
*/
createLine(N, Elem) :-
    length(Elem, N),
    maplist(random(0,2), Elem).

/*
    Creates a solution of only open doors (for puzzle
drawing purposes)
*/
createEmptySolution(Sol, N) :-
    NewN is N - 1,
    length(Sol, N),
    maplist(createEmptyLine(NewN), Sol).

/*
    Fills a certain line of the solution with 0's (open
doors)
*/
createEmptyLine(N, Elem) :-
    length(Elem, N),
    maplist(random(0,1), Elem).

/*
    Creates a random solution for the puzzle
```

```
*/
createGrid(Lines,Columns, N):-
    createSolution(Lines,N),
    createSolution(Columns,N).

/*
    Creates an empty solution for the puzzle (for drawing
purposes)
*/
createEmptyGrid(Lines,Columns, N):-
    createEmptySolution(Lines,N),
    createEmptySolution(Columns,N).

/*
    Iterates over puzzle (an empty puzzle) to calculate
the value of each room
*/
fillPuzzle(Puzzle, _, _, _, Size, Size, Puzzle):- !.
fillPuzzle(Puzzle, Lines, Columns, Size, Y, Size, Re-
turn):- !,
    NewY is Y + 1,
    fillPuzzle(Puzzle, Lines, Columns, 0, NewY, Size, Re-
turn).
fillPuzzle(Puzzle, Lines, Columns, X, Y, Size, Return):-
    calculateValue(Lines, Columns, X, Y, Value),
    replace(Puzzle, X, Y, Value, NewPuzzle),
    NewX is X + 1,
    fillPuzzle(NewPuzzle, Lines, Columns, NewX, Y, Size,
Return).

/*
    Calculates the value of a certain room, by finding
the first closed door in each direction
*/
calculateValue(Lines, Columns, X, Y, Value):-
    nth0(X, Lines, Line),
    nth0(Y, Columns, Column),
    divide(Line, LeftAux, Right, Y),
    divide(Column, UpAux, Down, X),
    reverse(LeftAux,Left),
    reverse(UpAux, Up),
    if_then_else( nth0(ValLeft, Left, 1), true,
length(Left,ValLeft)),
    if_then_else( nth0(ValRight, Right, 1), true,
length(Right,ValRight)),
```

```
    if_then_else( nth0(ValUp, Up, 1), true,
length(Up,ValUp)),
    if_then_else( nth0(ValDown, Down, 1), true,
length(Down,ValDown)),
    Value is ValLeft + ValRight + ValUp + ValDown + 1.

/*
    Creates a puzzle with size Size and all values = 0,
so that it can be latter filled with a solution
*/
createEmptyPuzzle(Size,Puzzle):-
    length(Puzzle,Size),
    maplist(createEmptyLine(Size), Puzzle).

/*
    Creates a puzzle with size Size, returning it in Puz-
zle and its solution in Lines (matrix of horizontal
    doors) and Columns (matrix of vertival doors)
*/
createPuzzle(Size, Puzzle, Lines, Columns):-
    createGrid(Lines,Columns, Size),
    createEmptyPuzzle(Size,EmptyPuzzle),
    fillPuzzle(EmptyPuzzle, Lines, Columns, 0, 0, Size,
Puzzle).
```

display.pl

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

/*
    Draws Up and Down limits of the puzzle according to
the Size
*/
drawLimit(0):- !, nl.
drawLimit(Size):-
    write('----'),
    NextSize is Size - 1,
    drawLimit(NextSize).
drawLimits(Size):-
    write(' ---'),
    Next is Size - 1,
    drawLimit(Next).
```

```
/*
    Iterates over puzzle, drawing at the same time closed
doors and puzzle numbers

    If the puzzle is to be shown unsolved, a solution of
only opens doors (0) should be passed
    as argument (i.e. in Lines and Columns)
*/
drawPuzzle(Puzzle, Lines, _, X, X) :- !,
    nth1(X, Puzzle, Number),
    nth1(X, Lines, Line),
    write('|'),
    drawNumber(Number, Line, X, 1).
drawPuzzle(Puzzle, Lines, Columns, Size, X) :-
    nth1(X, Puzzle, Number),
    nth1(X, Lines, Line),
    write('|'),
    drawNumber(Number, Line, Size, 1),
    getLine(Columns, X, Column),
    write('|'),
    drawLine(Column),
    NextX is X + 1,
    drawPuzzle(Puzzle, Lines, Columns, Size, NextX).
drawPuzzle(Puzzle, Lines, Columns, Size) :-
    drawLimits(Size),
    drawPuzzle(Puzzle, Lines, Columns, Size, 1),
    drawLimits(Size),
    nl, nl.

/*
    Draws a line of the puzzle, alternating puzzle num-
bers an possible closed doors
*/
drawNumber(Number, _, Y, Y) :-
    nth1(Y, Number, N),
    write(' '), write(N), write(' |'),
    nl.
drawNumber(Number, Line, Size, Y) :-
    nth1(Y, Number, N),
    write(' '), write(N), write(' '),
    nth1(Y, Line, Door),
    if_then_else(Door =:= 1, write('|'), write(' ')),
    NewY is Y + 1,
    drawNumber(Number, Line, Size, NewY).
```

```
/*
    Finds the Xth line of doors in Columns
*/
getLine(Columns, X, Column) :-
    applyToList(Columns, nth1, X, Column).

/*
    Draws a line of vertical doors between lines
*/
drawLine([]) :- write('|'), nl.
drawLine([C|Column]) :-
    if_then_else(C =:= 1, write('---'), write('   ')),
    length(Column, RestLength),
    if_then_else(RestLength > 0, write(' '), write('')),
    drawLine(Column).
```

solve.pl

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

/*
    Solves the puzzle Puzzle with size N, placing the so-
lution in Sol

    Sol is not in the same format as he solutions used
for display

    Example:
    Size = 4.

    Lines = [[A, B, C],
             [D, E, F],
             [G, H, I],
             [J, K, L]].
    Cols  = [[M, N, O],
             [P, Q, R],
             [S, T, U],
             [V, W, X]].

    Sol = [A, B, C, D, E, F, G, H, I, J, K, L, M, N, O,
P, Q, R, S, T, U, V, W, X].
*/
solvePuzzle(Puzzle, Sol,  N) :-
```

```
    length(Lines, N),
    length(Cols, N),
    NDoors is N - 1,
    fixMatrixColumns(Lines, NDoors),
    fixMatrixColumns(Cols, NDoors),
    solveNumbers(Puzzle, Lines, Cols, 0, 0, N),
    matrixToList(Lines, L),
    matrixToList(Cols, C),
    append(L, C, Sol),
    domain(Sol, 0, 1),
    labeling([], Sol).


/*
    Forces all lines of matrix to be of size Size
*/
fixMatrixColumns([], _).
fixMatrixColumns([Line|Matrix], Size) :-
    length(Line, Size),
    domain(Line, 0, 1),
    fixMatrixColumns(Matrix, Size).


/*
    Iterates over puzzle, adding the necessary re-
strictions to the solution depending on the value
    of the room
*/
solveNumbers(_, _, _, _, Size, Size):- !.
solveNumbers(Puzzle, Lines, Columns, Size, Y, Size):- !,
    NewY is Y + 1,
    solveNumbers(Puzzle, Lines, Columns, 0, NewY, Size).
    solveNumbers(Puzzle, Lines, Columns, X, Y, Size):-
    calculateDoors(Puzzle, Lines, Columns, X, Y, Size),
    NewX is X + 1,
    solveNumbers(Puzzle, Lines, Columns, NewX, Y, Size).

/*
    Adds restrictions to the solution using the value of
a certain room, by searching in all for directions
    for possible places for doors
*/
calculateDoors(Puzzle, Lines, Columns, X, Y, N) :-
    nth0(X, Lines, Line),
    nth0(Y, Columns, Column),
```

```
    matrix(Puzzle, X, Y, Value),
    Length is N - 1,
    searchLeft(Line, Y, ValLeft),
    searchLeft(Column, X, ValUp),
    searchRight(Line, Length, Y, ValRight),
    searchRight(Column, Length, X, ValDown),
    ValLeft + ValRight + ValUp + ValDown #= Value - 1.

/*
    Search for a door left of N in List
    (i.e. search for a door left in Lines and up in Col-
umns)
*/
searchLeft(_, 0, Val) :- !, Val #= 0.
searchLeft(List, N, Val) :-
    NextN is N - 1,
    searchLeft(List, NextN, NewVal),
    element(N, List, Elem),
    (Elem #= 1 #/\
    Val #= 0) #\/
    (Elem #= 0 #/\
    Val #= 1 + NewVal).

/*
    Search for a door right of N in List
    (i.e. search for a door right in Lines and down in
Columns)
*/
searchRight(_, N, N, Val) :- !, Val #= 0.
searchRight(List, Length, N, Val) :-
    NextN is N + 1,
    searchRight(List, Length, NextN, NewVal),
    Index is N + 1,
    element(Index, List, Elem),
    (Elem #= 1 #/\
    Val #= 0) #\/
    (Elem #= 0 #/\
    Val #= 1 + NewVal).
```

utility.pl

```
/* Implements an if-then-else-like decision stucture
```

```
    if(Condition)
    {
        Action1
    }
    else
    {
        Action2
    }
*/
if_then_else(Condition, Action1, _):- Condition, !, Ac-
tion1.
if_then_else(_,_,Action2):- Action2.

/*
    For each element (L1) of the first list creates the
element of the second list (R1), such that
        Pred(Arg, L1, R1)
    is true.
*/
applyToList([], _, _, []).
applyToList([L1|L], Pred, Arg, [R1|R]) :-
    O =.. [Pred, Arg, L1, R1],
    O,
    applyToList(L, Pred, Arg, R).

/*
    True if the element with row I and column J of Matrix
is Value
*/
matrix(Matrix, I, J, Value) :-
    nth0(I, Matrix, Row),
    nth0(J, Row, Value).

/*
    Divides a list into two lists using Index as a divid-
ing point

    Example:
        divide([1, 2, 3, 4], L1, L2, 1).
        L1 = [1],
        L2 = [2, 3, 4] ?
*/
divide(List, L1, L2, Index):-
    length(List, Length),
    AuxLength is Length - Index,
```

```
    append(L1, L2, List),
    length(L1, Index),
    length(L2, AuxLength).

/*
    True if Matrix and NewMatrix are composed of equal
elements, except for the element of row X and column
    Y, which has NewVal value

    i.e., replaces the element of row X and col Y in Ma-
trix with NewVal
*/
replace( Matrix , X , Y , NewVal , NewMatrix ) :-
    append(RowPfx,[Row|RowSfx],Matrix),
    length(RowPfx,X),
    append(ColPfx,[_|ColSfx],Row),
    length(ColPfx,Y),
    append(ColPfx,[NewVal|ColSfx],RowNew),
    append(RowPfx,[RowNew|RowSfx],NewMatrix).

/*
    Converts a matrix to a list, appending each line to
the next

    Example:
        matrixToList([[1, 2, 3], [4, 5, 6], [7, 8, 9]],
M).
        M = [1, 2, 3, 4, 5, 6, 7, 8, 9] ?
*/
matrixToList([], []).
matrixToList([[] | Matrix], List) :-
    matrixToList(Matrix, List).
matrixToList([[L|L1] | Matrix], [L|List]) :-
    matrixToList([L1|Matrix], List).

/*
    Converts a list to a matrix with NRows rows and NCols
columns, assuming each line is followed by
    the next one

    Example:
        listToMatrix([1, 2, 3, 4, 5, 6], L, 2, 3).
        M = [[1, 2, 3], [4, 5, 6]] ?
*/
listToMatrix(List, Matrix, NRows, NCols) :-
```

```
    listToMatrix(List, Matrix, 0, 0, NRows, NCols).

listToMatrix([], [], NRows, _, NRows, _).
listToMatrix(List, [[]|Matrix], Row, NCols, NRows, NCols)
:-
    NextRow is Row + 1,
    listToMatrix(List, Matrix, NextRow, 0, NRows, NCols).
listToMatrix([L|List], [[L|L1]|Matrix], Row, Col, NRows,
NCols) :-
    NextCol is Col + 1,
    listToMatrix(List, [L1|Matrix], Row, NextCol, NRows,
NCols).
```

8.2 Exemplos de Execução da Aplicação

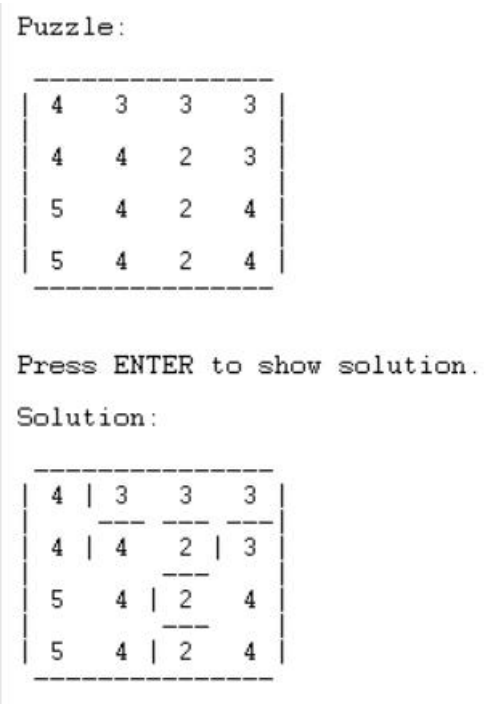


Fig. 4. Exemplo de execução do predicado *doors* com argumento 4

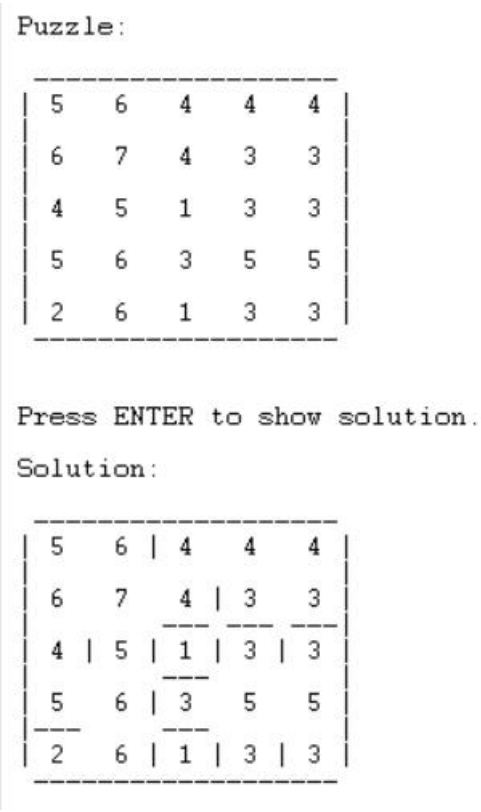


Fig. 5. Exemplo de execução do predicado *doors* com argumento 5

```
Puzzle:

 ----------------------
| 3   1   1   5   6   3 |
|                       |
| 3   4   3   6   7   5 |
|                       |
| 4   4   2   6   7   5 |
|                       |
| 3   4   4   4   5   3 |
|                       |
| 3   1   3   2   6   4 |
|                       |
| 3   1   3   2   2   4 |
 ----------------------

Press ENTER to show solution.

Solution:

 ----------------------
| 3 | 1 | 1 | 5   6 | 3 |
|     ---   ---         |
| 3 | 4   3 | 6   7   5 |
|               |       |
| 4   4 | 2 | 6   7   5 |
|---        ---       ---|
| 3 | 4   4 | 4 | 5 | 3 |
|     ---       ---     |
| 3 | 1 | 3 | 2 | 6   4 |
|     ---           ---   |
| 3 | 1 | 3 | 2 | 2   4 |
 ----------------------
```

Fig. 6. Exemplo de execução do predicado *doors* com argumento 6

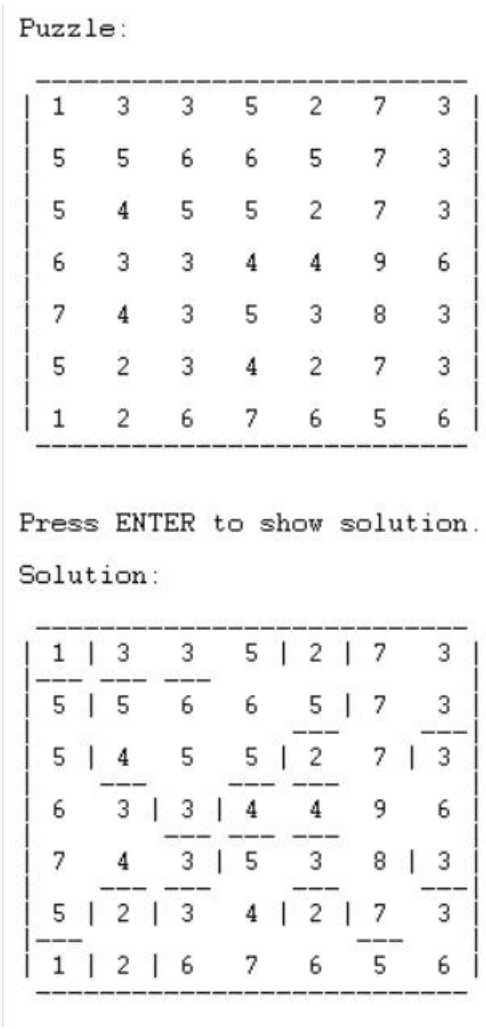


Fig. 7. Exemplo de execução do predicado *doors* com argumento 7

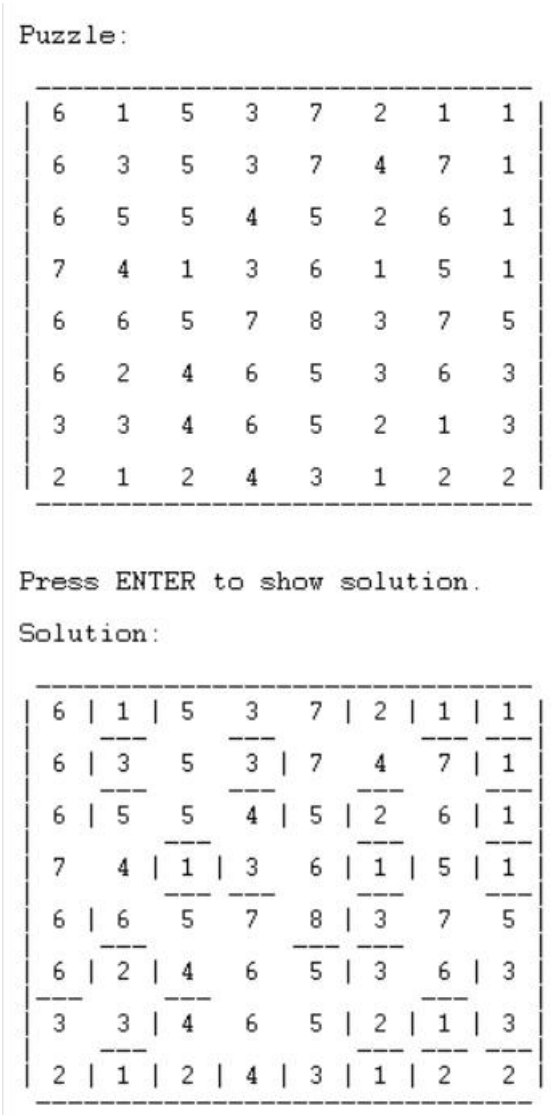


Fig. 8. Exemplo de execução do predicado *doors* com argumento 8