# DualClue Cross-a-Pix

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

Flávio Couto and Pedro Afonso Castro

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

**Resumo** Este projeto, realizado no âmbito da unidade curricular de Programação em Lógica, vem ajudar a consolidar os conhecimentos mais avançados de Prolog adquiridos deste o primeiro projeto, acerca de Programação em Lógica com restrições, consistindo em escrever um programa Prolog capaz de resolver qualquer puzzle do tipo "DualClue Cross-a-Pix". Este artigo tem então como objetivo complementar e descrever o projeto desenvolvido.

## 1 Introdução

Este trabalho tem como objetivo adquirir compentências ao nível da programação em lógica, servindo de continuação ao primeiro trabalho realizado na primeira metade do semestre. Neste projeto, foram-nos propostos vários puzzles e problemas de otimização, que deveriam ser resolvidos através de restrições. Para além disso, no caso de ser escolhido um puzzle (como é o caso do nosso grupo), o programa deve também ser capaz de gerar um puzzle aleatório. O nosso grupo resolveu escolher o puzzle DualClue Cross-a-Pix. O puzzle consiste numa matriz M x N, constituída por várias regiões, em que cada secção deve estar pintada ou não, de acordo com indicações dadas para cada linha e coluna. Este artigo descreve então o puzzle DualClue Cross-a-pix, a abordagem utilizada para resolver qualquer puzzle, desde que o mesmo tenha uma solução, a visualização desta no momento em que esta é gerada, análise de exemplos de aplicação em instâncias do programa e, por último, as conclusões que retiramos deste projeto.

## 2 Descrição do Problema

O DualClue Cross-a-Pix é um puzzle que consiste numa matriz M x N, dividida em várias regiões, com duas pistas em cada linha e coluna - a primeira representa o número de quadrados que estão pintados ao longo dessa linha/coluna, e a segunda representa o número de blocos (ou seja, o número de secções de quadrados pintados consecutivamente, como se pode ver na Fig. 3) ao longo dessa linha/coluna. Todos os quadrados de uma secção devem estar ou pintados ou não pintados.

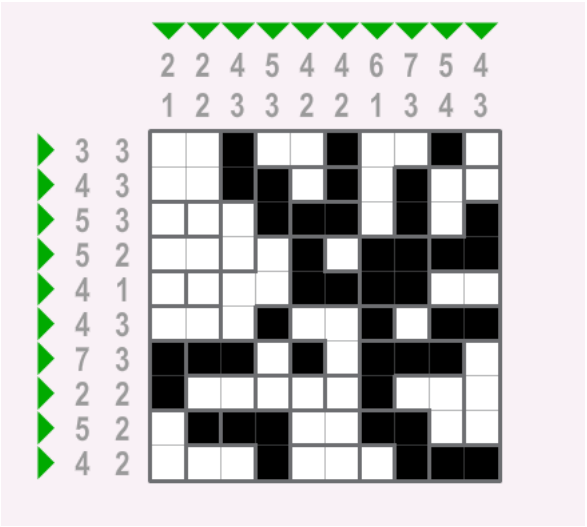


**Figura 1.** Puzzle DualClue Cross-a-Pix depois de resolvido

## 3 Abordagem

Na implementação do puzzle em Prolog, optámos por representar os dados em várias estruturas de dados:

- Uma lista de listas, de tamanho M x N-1 em que cada elemento toma o valor 0 ou 1, significando 1 ter uma parede à sua direita e 0 a sua ausência (define-se uma parede como sendo a delimitação de uma região) (definida daqui para a frente como **HP**).
- Outra lista de listas com tamanho N x M-1, que guarda a mesma informação, mas para a existência ou não de parede em baixo (definida daqui para a frente como **VP**).
- Uma lista de pares, em que cada par representa os números acima de cada coluna (definida daqui para a frente como **C**).
- Uma lista de pares, em que cada par representa os números acima de cada linha (definida daqui para a frente como **L**).

### 3.1 Variáveis de Decisão

A solução é dada numa lista de listas (daqui para a frente referida como **T**). Cada lista contém informação sobre cada linha, sendo o domínio de cada lista 0,1, sendo que 0 representa um quadrado não pintado, e 1 um quadrado pintado. O tamanho de T é, então M x N.

Apresentamos de seguida o predicado que determina a solução:

```
doubleCrossAPixSolver(-HP, -VP, +T, -V, -H) :-
```

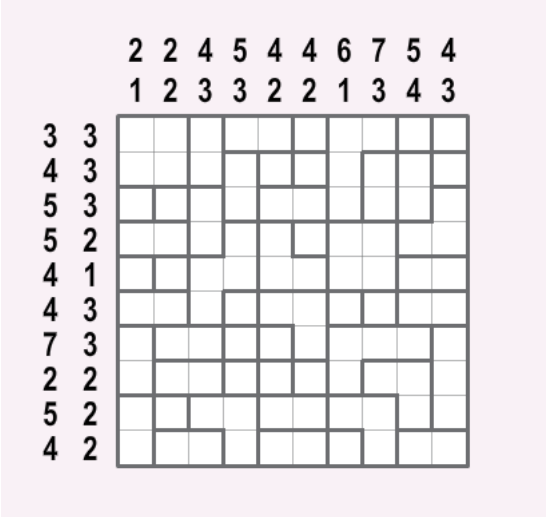


**Figura 2.** Estado do puzzle antes de ser começado. Podemos ver no topo e à esquerda as pistas dadas ao jogador

```
append(T, Vars),
domain(Vars, 0, 1),
groupsWithSameColour(HP,VP,T),
horizontalRule(T,H),
verticalRule(T,V),
labeling([ff,enum],Vars).
```

### 3.2 Restrições

O puzzle pode ser resolvido recorrendo a 5 restrições:

1. Para cada elemento pertencente ao mesmo bloco, a sua cor tem de ser igual;
2. O numero de quadrados pretos de uma coluna C tem de ser igual a V[C][0];
3. O numero de blocos pretos de uma coluna C tem de ser igual a V[C][1];
4. O numero de quadrados pretos de uma linha L tem de ser igual a H[L][0];
5. O numero de blocos pretos de uma linha L tem de ser igual a H[L][1].

**Para cada elemento pertencente ao mesmo bloco, a sua cor tem de ser igual.** Para cada elemento do puzzle, são testados todos os elementos na vertical e horizontal até encontrarmos uma parede - ou seja, fim do bloco. Verifica-se se têm todos a mesma cor. Esta restrição é testada no seguinte predicado:

```
groupsWithSameColour(-HP, -VP, +T).
```

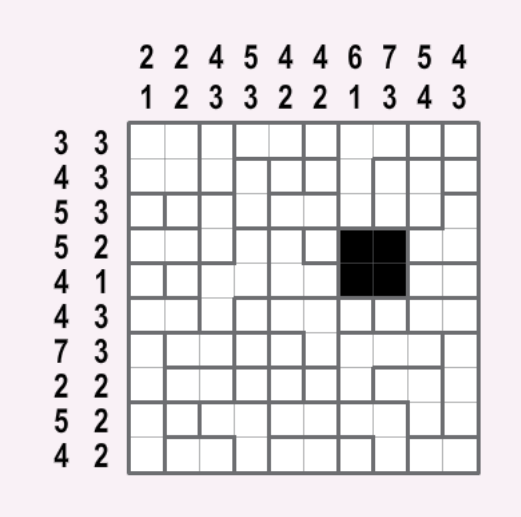


**Figura 3.** Uma secção

**O numero de quadrados pretos de uma coluna C tem de ser igual a V[C][0] e O numero de blocos pretos de uma coluna C tem de ser igual a V[C][1].**

Percorre-se a coluna C e contam-se os quadrados pintados e as secções. Verifica-se se são iguais a V[C][0] e V[C][1], respetivamente. Estas duas restrições são testadas no mesmo predicado, sendo este o seguinte:

```
verticalRule(-T,+V).
```

**O numero de quadrados pretos de uma coluna C tem de ser igual a V[C][0] e O numero de blocos pretos de uma coluna C tem de ser igual a V[C][1].**

Percorre-se a linha L e contam-se os quadrados pintados e as secções. Verifica-se se são iguais a H[L][0] e H[L][1], respetivamente. Estas duas restrições são testadas no mesmo predicado, sendo este o seguinte:

```
horizontalRule(-T,+H).
```

### 3.3 Estratégia de Pesquisa

No que toca ao *labeling*, usamos as seguintes estratégias:

- Escolha de variável: estratégia 'ffc', que consiste na escolha da variável de menor domínio com maior número de restrições.
- Ordenação de valores do domínio: estratégia 'up', que seleciona os valores do domínio por ordem ascendente.
- Branching : estratégia 'enum', atribuindo sequencialmente os valores do domínio à variável.

## 4 Visualização da Solução

A solução do puzzle é mostrada no ecrã de forma simples, mas de fácil percepção. A Fig. 4 mostra o resultado após o programa resolver um puzzle.

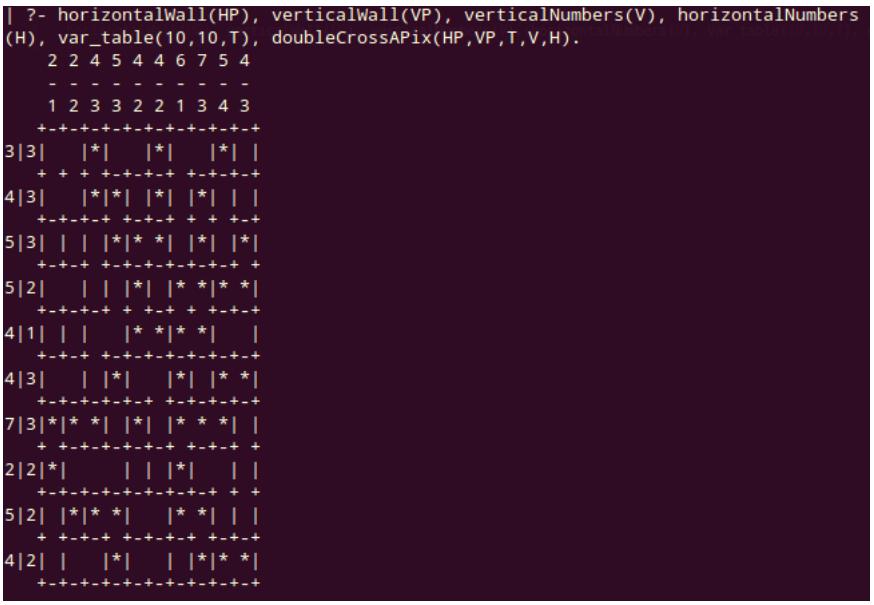


**Figura 4.** Output produzido no terminal para o puzzle

Para imprimir a solução, recorremos ao seguinte predicado:

`print_puzzle(+HP, +VP, -T, +H, +V)` .

Em que HP, VP, T, H e V têm os significados já descritos anteriormente.

## 5 Resultados

Foram realizados vários testes para avaliar a prestação do programa, sujeitando-o a várias condições, com complexidade diferente. Para avaliar a sua prestação, recorremos ao predicado statistics/2 para determinar o tempo que decorreu entre o início e o fim da execução do teste. É importante referir que a precisão deste predicado vai até ao centésimo de segundo.

Foram feitos dois testes, um para testar o tempo necessário para gerar um puzzle e outro para testar o tempo necessário para gerar um puzzle e resolvê-lo. Implementámos dois predicados para correr estes testes, sendo eles os seguintes:

```
assess_time_generator(-Tries, -M, -N, +Average).
assess_time_generate_and_solve(-Tries, -M, -N, +Average).
```

Apresenta-se então os testes realizados e seus resultados.

**Tabela 1.** Resultados do teste ao tempo necessário para gerar um puzzle

| Tamanho do Puzzle | Número de execuções | Tempo médio de execução |
|---|---|---|
| 5x5 | 300 | 0.0011s |
| 10x10 | 300 | 0.0022s |
| 20x20 | 300 | 0.0063s |
| 50x50 | 100 | 0.0385s |
| 100x100 | 100 | 0.1724s |

**Tabela 2.** Resultados do teste ao tempo necessário para gerar um puzzle e resolvê-lo

| Tamanho do Puzzle | Número de execuções | Tempo médio de execução |
|---|---|---|
| 5x5 | 100 | 0.0008s |
| 10x10 | 100 | 0.0063s |
| 50x50 | 100 | 0.0518s |
| 100x100 | 100 | 0.2143s |

Com isto, podemos tirar várias conclusões:

- O programa encontra soluções bastante rápido para puzzles com tamanho até 40x40. A partir daí, começa a demorar bastante tempo para conseguir resolver o puzzle. No entanto, tendo em conta que o número de células e regiões começa a entrar na ordem dos milhares, e as hipóteses a testar na ordem dos decaliões, tal resultado não é de estranhar.
- O tempo para resolver um puzzle é bastante superior ao tempo para gerar um puzzle aleatório.

– Tanto o tempo para gerar um puzzle, como o tempo para gerar um e resolvê-lo (de 2500 para 10000 células, houve um aumento de 4 vezes do tempo de execução em ambos os casos, e essa tendência é verificável também, por exemplo, de 100 para 400 células no caso da geração e resolução), aumenta de forma linear.

## 6 Conclusões

Contrariamente ao projeto anterior, não foi preciso tanto tempo para concluir este. No entanto, não é por isso que deixámos de cumprir o principal objetivo de consolidar os conhecimentos referentes à programação em lógica com restrições.

O uso de restrições não só vem facilitar bastante a resolução de problemas como o DualClue Cross-a-Pix, como o faz de forma extremamente rápida e eficiente. Depois de compreendido, o grupo considera este um conceito extremamente interessante de colocar em prática, e o facto de o puzzle ser bastante cativante tornou a realização deste projeto uma experiência bastante agradável.

## Anexo

### Código fonte

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

/* Predicados para testar e avaliar tempos de execucao */

assess_time_generator(Tries, M, N, Average) :-
        assess_time_generator_aux(Tries, M, N, Times),
        sumlist(Times, Sum),
        length(Times, Length),
        Average is Sum / Length.

assess_time_generator_aux(0, _, _, []).
assess_time_generator_aux(Tries, M,N, [Time|Times]) :-
        statistics(walltime, [Start|_]),
        test_generator(M, N),
        statistics(walltime, [End|_]),
        Time is End - Start,
        X is Tries - 1,
```

```
        assess_time_generator_aux(X, M, N, Times).

assess_time_generate_and_solve(Tries, M, N, Average) :-
        assess_time_generate_and_solve_aux(Tries, M, N,
            Times),
        sumlist(Times, Sum),
        length(Times, Length),
        Average is Sum / Length.

assess_time_generate_and_solve_aux(0, _, _, []).
assess_time_generate_and_solve_aux(Tries, M,N, [Time|
    Times]) :-
        statistics(walltime, [Start|_]),
        test_generator(M, N),
        statistics(walltime, [End|_]),
        Time is End - Start,
        X is Tries - 1,
        assess_time_generator_aux(X, M, N, Times).


teste :-
        horizontalWall(HP),
        verticalWall(VP),
        verticalNumbers(V),
        horizontalNumbers(H),
        length(V, NRows),
        length(H, NCols),
        doubleCrossAPix(HP, VP, NRows, NCols, V, H).

teste_print :-
        horizontalWall(HP),
        verticalWall(VP),
        verticalNumbers(V),
        horizontalNumbers(H),
        print_white_puzzle(HP, VP, H, V).

test_generator(M, N) :-
        game_generator(M, N, HP, VP, H, V),
        print_white_puzzle(HP, VP, H, V).

test_generate_and_solve(M, N) :-
        game_generator(M, N, HP, VP, H, V),
        print_white_puzzle(HP, VP, H, V),
        doubleCrossAPix(HP, VP, M, N, V, H).
```

```
horizontalWall(Walls) :- Walls = [
[0,1,1,0,1,1,0,1,1],
[0,1,1,1,1,1,1,1,1],
[1,1,1,1,0,1,1,1,1],
[0,1,1,1,1,1,0,1,0],
[1,1,0,1,0,1,0,1,0],
[0,1,1,1,0,1,1,1,0],
[1,0,1,1,1,1,0,0,1],
[1,0,0,1,1,1,1,0,1],
[1,1,0,1,0,1,0,1,1],
[1,0,1,1,0,1,1,1,0]
].

verticalWall(Walls) :- Walls =    [
[0,1,1,1,1,1,0,1,0],
[0,1,1,1,1,1,1,1,1],
[0,1,0,1,0,1,1,1,1],
[1,0,1,0,1,1,1,1,0],
[1,1,1,0,1,1,1,1,1],
[1,1,1,1,1,0,1,1,1],
[0,0,1,0,1,1,0,1,1],
[1,0,1,0,1,1,1,1,0],
[1,0,1,1,1,1,1,0,1],
[1,1,0,1,1,1,0,0,1]].

verticalNumbers(Numbers) :- Numbers = [
[2,1],
[2,2],
[4,3],
[5,3],
[4,2],
[4,2],
[6,1],
[7,3],
[5,4],
[4,3]
].

horizontalNumbers(Numbers) :- Numbers = [
[3,3],
[4,3],
[5,3],
[5,2],
[4,1],
```

```
[4,3],
[7,3],
[2,2],
[5,2],
[4,2]
].

var_table(0, _, []) :- !.

var_table(M, N, T) :-
        M > 0, !,
        length(Row, N),
        M1 is M - 1,
        var_table(M1, N, T1),
        append([Row], T1, T).

doubleCrossAPix(HP, VP, T, V, H) :-
        doubleCrossAPixSolver(HP, VP, T, V, H),
        print_puzzle(HP, VP, T, H, V) .

doubleCrossAPix(HP, VP, NRows, NCols, V, H) :-
        var_table(NRows, NCols, T),
        doubleCrossAPix(HP, VP, T, V, H).

doubleCrossAPixSolver(HP, VP, T, V, H) :-
        append(T, Vars),
        domain(Vars, 0, 1),
        groupsWithSameColour(HP,VP,T),
        horizontalRule(T,H),
        verticalRule(T,V),
        labeling([ffd,enum],Vars).

/* Para cada elemento pertencente ao mesmo bloco, a sua
   cor tem de ser igual */

groupsWithSameColour(HP, VP, T) :-
        groupsWithSameColourHorizontal(HP, T),
        groupsWithSameColourVertical(VP, T).

groupsWithSameColourVertical(VP, T) :-
        transpose(T, TT),
        groupsWithSameColourHorizontal(VP, TT).

groupsWithSameColourHorizontal([],[]).
```

```
groupsWithSameColourHorizontal([HRowWalls|HRowsWalls],[
    Row|Rows]) :-
        checkWalls(HRowWalls, Row),
        groupsWithSameColourHorizontal(HRowsWalls, Rows).

checkWalls([], [_]). /* Last elem */
checkWalls([HWall|HWalls], [Elem1,Elem2|Elems]) :-
        checkWall(HWall, Elem1, Elem2),
        checkWalls(HWalls, [Elem2|Elems]).

checkWall(0, Elem1, Elem2) :-
        Elem1 #= Elem2.
checkWall(1, _, _).

/* O numero de quadrados pretos da coluna C tem de ser
    igual a V[C][0]
        O numero de blocos pretos da coluna C tem de ser
            igual a V[C][1]  */

horizontalRule([], []).
horizontalRule([Row|Rows],[Rule|Rules]) :-
        validate_row(Row, Rule), horizontalRule(Rows,
            Rules).

validate_row(Row, [PaintedSquares,NumberSections]) :-
        check_row_painted_squares(Row, PaintedSquares),
        check_row_sections(Row, NumberSections).

check_row_painted_squares([], 0).
check_row_painted_squares([Elem|Row], PaintedSquares) :-
        Elem #= 0,
        check_row_painted_squares(Row, PaintedSquares).
check_row_painted_squares([Elem|Row], PaintedSquares) :-
        Elem #= 1,
        N is PaintedSquares - 1,
            check_row_painted_squares(Row, N).

check_row_sections([Elem|Row], NumberSections) :- Elem #=
    0,
        check_row_sections_aux(Row, 0, 0, NumberSections)
            .
check_row_sections([Elem|Row], NumberSections) :- Elem #=
    1,
        check_row_sections_aux(Row, 1, 1, NumberSections)
            .
```

```
check_row_sections_aux([], _, N, N).
check_row_sections_aux([Elem|Row], LastElem,
    CurrentNumberSections, NumberSections) :-
        Elem #= 1,
        LastElem #= 0,
        !,
        N is CurrentNumberSections + 1,
        check_row_sections_aux(Row, Elem, N,
            NumberSections).
check_row_sections_aux([Elem|Row], _,
    CurrentNumberSections, NumberSections) :-
        check_row_sections_aux(Row, Elem,
            CurrentNumberSections, NumberSections).

/*      O numero de quadrados pretos da linha L tem de
    ser igual a V[H][0]
        O numero de blocos pretos da linha L tem de ser
            igual a V[H][1] */

verticalRule(Rows, Rules) :-
        transpose(Rows, Cols), horizontalRule(Cols, Rules
            ).

/* Imprimir solucao em formato legivel */
print_white_puzzle(HP, VP, H, V) :-
        length(H, NumRows),
        length(V, NumCols),
        white_board(NumRows, NumCols, T),
        print_puzzle(HP, VP, T, H, V).

white_board(0, _, []).
white_board(NumRows, NumCols, [Row|Rows]) :-
        NumRows > 0,
        white_row(NumCols, Row),
        X is NumRows - 1,
        white_board(X, NumCols, Rows).

white_row(0, []).
white_row(NumCols, [0|Cols]) :-
        NumCols > 0,
        X is NumCols - 1,
        white_row(X, Cols).

print_puzzle(HP, VP, T, H, V) :-
```

```
        getSpacing(H, PaintedSpace, SectionsSpace),
        Spacing is PaintedSpace + 1 + SectionsSpace,
        length(V, NumCols),
        print_vertical_rules(V, Spacing),
        transpose(VP, VPT),
        print_horizontal_border(Spacing, NumCols),
        print_puzzle_aux(HP, VPT, T, H, PaintedSpace,
            SectionsSpace, Spacing).

print_vertical_rules(V, Spacing) :-
        getSpacing(V, PaintedSpace, SectionsSpace),
        transpose(V, [Painted, Sections]),
        print_vertical_numbers(Spacing, Painted,
            PaintedSpace),
        length(Painted, NumCols),
        print_division(Spacing, NumCols),
        print_vertical_numbers(Spacing, Sections,
            SectionsSpace).

print_vertical_numbers(_, _, 0).

print_vertical_numbers(Spacing, Numbers, NumbersSpace) :-
        print_spacing(Spacing),
        N is NumbersSpace - 1,
        print_vertical_numbers_aux(Numbers, N, Remainders
            ),
        print_vertical_numbers(Spacing, Remainders, N).

print_vertical_numbers_aux([], _, []) :-
        write('\n').
print_vertical_numbers_aux([Number|Numbers], N, [
    Remainder|Remainders]) :-
        Digit is Number // 10 ^ N,
        Remainder is Number rem 10 ^ N,
        write('␣'),
        write(Digit),
        print_vertical_numbers_aux(Numbers, N, Remainders
            ).

print_division(Spacing, NumCols) :-
        print_spacing(Spacing),
        print_division_aux(NumCols).

print_division_aux(0) :-
        write('\n').
```

```
print_division_aux(NumCols) :-
        N is NumCols - 1,
        write('-'),
        print_division_aux(N).

print_horizontal_border(Spacing, NumCols) :-
        print_spacing(Spacing),
        print_line(NumCols),
        write('\n').

getSpacing(H, PaintedSpace, SectionsSpace) :-
        transpose(H, [Painted, Sections]),
        max_member(MaxPainted, Painted),
        PaintedSpace is truncate(log(10,MaxPainted)) + 1,
        max_member(MaxSections, Sections),
        SectionsSpace is truncate(log(10,MaxSections)) +
            1.

print_spacing(0).
print_spacing(Spacing) :-
        X is Spacing - 1,
        write(' '),
        print_spacing(X).

print_line(0) :-
        write('+').
print_line(NumCols) :-
        X is NumCols - 1,
        write('+-'),
        print_line(X).


print_puzzle_aux([HWall], [], [Row], [HRule],
    PaintedSpace, SectionsSpace, Spacing) :-
        print_horizontal_rule(HRule, PaintedSpace,
            SectionsSpace),
        print_row(HWall, Row),
        write('\n'),
        length(Row, NumCols),
        print_horizontal_border(Spacing, NumCols),
        write('\n').

print_puzzle_aux([HWall|HWalls], [VWall|VWalls], [Row|
    Rows], [HRule|HRules], PaintedSpace, SectionsSpace,
    Spacing) :-
```

```
        print_horizontal_rule(HRule, PaintedSpace,
            SectionsSpace),
        print_row(HWall, Row),
        write('\n'),
        print_vertical_walls(VWall, Spacing),
        write('\n'),
        print_puzzle_aux(HWalls, VWalls, Rows, HRules,
            PaintedSpace, SectionsSpace, Spacing).

print_horizontal_rule([Painted,Sections], PaintedSpace,
    SectionsSpace) :-
        print_number_horizontal(Painted, PaintedSpace),
        write('|'),
        print_number_horizontal(Sections, SectionsSpace).

print_number_horizontal(_, 0).
print_number_horizontal(Number, Space) :-
        N is Space - 1,
        Digit is Number // 10 ^ N,
        Remainder is Number rem 10 ^ N,
        write(Digit),
        print_number_horizontal(Remainder, N).

print_row(Walls, Elems) :-
        write('|'),
        print_row_aux(Walls, Elems).

print_row_aux([], [Elem]) :-
        print_elem(Elem),
        write('|').

print_row_aux([Wall|Walls], [Elem|Elems]) :-
        print_elem(Elem),
        print_horizontal_wall(Wall),
        print_row_aux(Walls, Elems).

print_elem(0) :-
        write('␣').
print_elem(1) :-
        write('*').

print_horizontal_wall(0) :-
        write('␣').
print_horizontal_wall(1) :-
        write('|').
```

```
print_vertical_walls(VWall, Spacing) :-
        print_spacing(Spacing),
        print_vertical_walls_aux(VWall).

print_vertical_walls_aux([]) :-
        write('+').

print_vertical_walls_aux([VWall|VWalls]) :-
        write('+'),
        print_vertical_wall(VWall),
        print_vertical_walls_aux(VWalls).

print_vertical_wall(0) :-
        write('␣').
print_vertical_wall(1) :-
        write('-').

/* Gerar um tabuleiro aleatorio */

/* Gerar uma matriz de informacoes referentes as paredes
   de tamanho M*N */
generate_walls(0,_,[]) :- !.
generate_walls(M,N,[Row|T]) :- M > 0, !,
   generate_line_walls(N,Row), M1 is M-1, generate_walls(
   M1,N,T).
generate_line_walls(0, []) :- !.
generate_line_walls(N,[Elem|Row]) :- N > 0, !, N1 is N-1,
    random(0,2,Elem), generate_line_walls(N1, Row).

/* Gerar as duas matrizes com informacao referente aos
   quadrados pintados / seccoes */
get_horizontal_numbers([], []).
get_horizontal_numbers([Row|Rows], [[PaintedSquares,
   NumberSections]|NumbersRows]) :-
        get_row_numbers(Row, PaintedSquares),
        get_row_sections(Row, NumberSections),
        get_horizontal_numbers(Rows, NumbersRows).

get_row_numbers(Row, PaintedSquares) :-
        sumlist(Row, PaintedSquares).

get_row_sections([Elem|Row], NumberSections) :-
        get_row_sections_aux(Row, Elem, Elem,
            NumberSections).
```

```
get_row_sections_aux([], _, N, N).
get_row_sections_aux([1|Row], 0, CurrentNumberSections,
    NumberSections) :-
        N is CurrentNumberSections + 1,
        get_row_sections_aux(Row, 1, N, NumberSections).
get_row_sections_aux([Elem|Row], _,CurrentNumberSections,
     NumberSections) :-
        get_row_sections_aux(Row, Elem,
            CurrentNumberSections, NumberSections).

get_vertical_numbers(Rows, Rules) :-
        transpose(Rows, Cols), get_horizontal_numbers(
            Cols, Rules).

/* Gerar um jogo M por N */
game_generator(M, N, HP, VP, H, V) :-
        generate_walls(M, N, HP, VP),
        var_table(M, N, T),
        paint_board(HP, VP, T),
        calculateRules(T, H, V).

generate_walls(M, N, HP, VP) :-
        M1 is M - 1,
        N1 is N - 1,
        generate_walls(M, N1, HP),
        generate_walls(N, M1, VP).

paint_board(HP, VP, T) :-
        append(T, Vars),
        domain(Vars, 0, 1),
        groupsWithSameColour(HP,VP,T),
        restrainRandomPositions(T),
        labeling([ff,enum],Vars).

calculateRules(T, H, V) :-
        get_horizontal_numbers(T, H),
        get_vertical_numbers(T,V).

restrainRandomPositions(T) :-
        getBoardSize(T, NumRows, NumCols),
        MinElems is truncate(sqrt(min(NumRows, NumCols)))
            ,
        MaxElems is truncate(sqrt(NumRows * NumCols)) +
            1,
        random(MinElems, MaxElems, NumElemsToPaint),
```

```
        getRandomPositions(NumElemsToPaint, NumRows,
            NumCols, Positions),
        restrainElems(Positions, T).

getBoardSize(T, NumRows, NumCols) :-
        length(T, NumRows),
        nth0(0, T, Row),
        length(Row, NumCols).

getRandomPositions(NumPositions, NumRows, NumCols,
    Positions) :-
        getRandomPositionsAux(NumPositions, NumRows,
            NumCols, PositionsList),
        remove_dups(PositionsList, Positions).
getRandomPositionsAux(0, _,_,[]) :- !.
getRandomPositionsAux(NumPositions, NumRows, NumCols, [[
    Row,Col]|Positions]) :-
        NumPositions > 0, !,
        random(0, NumRows, Row),
        random(0, NumCols, Col),
        X is NumPositions - 1,
        getRandomPositions(X, NumRows, NumCols, Positions
            ).

restrainElems([], _).
restrainElems([[Row,Col]|Positions], T) :-
        nth0(Row, T, TRow),
        nth0(Col, TRow, Elem),
        Elem #= 1,
        restrainElems(Positions, T).
```