

# Resolução de um problema de restrições

Alexandre José da Silva Carvalho - up201506688  
Vitor Emanuel Fernandes Magalhães - up201504818

FEUP-PLOG  
Turma 3MIEIC6  
Grupo JapaneseSums.5

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

**Resumo** Deve contextualizar e resumir o trabalho, salientando o objetivo, o método utilizado e fazendo referência aos principais resultados e à principal conclusão que esses resultados permitem obter.

**Keywords:** Restrições, PROLOG, biblioteca CLP(FD)

## 1 Introdução

Os objetivos deste trabalho, proposto pelos docentes, envolvem a resolução de um problema de restrições, na forma de um jogo de tabuleiro, utilizando a linguagem PROLOG e a biblioteca **CLP(FD)**. O jogo de tabuleiro escolhido para este trabalho foi o Japanese Sums<sup>1</sup>.

O artigo seguirá a estrutura recomendada pelos docentes:

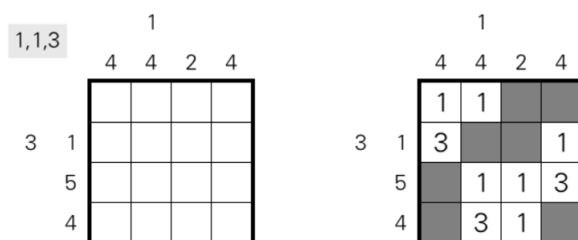
- **Introdução:** Introdução do trabalho;
- **Descrição do Problema:** Descrição detalhada do problema;
- **Abordagem:** Descrição da modelação do problema, indicando as Variáveis, os Domínios e as Restrições, assim como a Função de Avaliação e a Estratégia de Pesquisa;
- **Visualização da Solução:** Explicação dos predicados que permitem a visualização dos tabuleiros;
- **Resultados:** Exemplos de resultados com diferentes complexidades;
- **Conclusões:** Conclusões obtidas;
- **Bibliografia:** Bibliografia consultada;
- **Anexos:** Anexos pedidos.

## 2 Descrição do Problema

O problema de decisão escolhido foi o puzzle Japanese Sums.

Este puzzle, semelhante a jogos como *Kakuro* e até ao próprio *Sudoku*, baseia-se no preenchimento de células de um tabuleiro, de maneira que a soma de cada

linha e coluna respectiva seja igual ao valor indicado nessa linha ou coluna. Algumas células estão preenchidas por uma célula cinzenta, que divide a linha/coluna em várias secções. Para estes casos, são indicados valores que têm de ser correspondidos em cada secção.



**Figura 1.** Exemplo de um tabuleiro preenchido

Neste exemplo, a segunda linha contém dois números, 3 e 1, que correspondem a cada soma de cada secção dessa linha.

### 3 Abordagem

Como sugerido pelos docentes, a abordagem do problema foi feita tendo em conta um *PSR*, ou seja, um Problema de Satisfação de Restrições, utilizando a linguagem PROLOG.

Um *PSR* é modelado através das variáveis, dos seus domínios e das restrições utilizadas.

#### 3.1 Variáveis de Decisão

Para resolver o problema, começou-se por definir as variáveis de decisão.

Cada elemento da matriz de solução foi definido como uma variável de decisão, com domínio entre 0 e 9.

De seguida, foi decidido que cada elemento da linha tem de estar entre 0 e o valor máximo das somas daquela lista.

As células escurecidas foram definidas e representadas com valor 0 para um cálculo mais acessível.

#### 3.2 Restrições

Após as variáveis de decisão e os seus domínios estarem definidos, implementaram-se as restrições.

A primeira restrição tem como objetivo asseverar que as somas da linha a ser analisada são iguais à soma dos elementos da lista de somas dessa linha. Foi utilizado o predicado *sum(+Vars, +Rel, ?Expr)*.

De seguida, é feita uma verificação ao tamanho da lista de somas relativa à linha a analisar:

- **Se entre 0 e 1**: restringe a linha, garantindo que só haverá 0 ou 1 elementos na linha, sendo o resto preenchido por células sombreadas;
- **Se maior ou igual a 2**: restringe a linha, de maneira a garantir as divisões da linha.

Nestas restrições, foram utilizadas restrições materializáveis.

### 3.3 Função de Avaliação

Como argumentos para o início do programa, são pedidas as somas das linhas e as somas das colunas. O programa é feito à volta destas listas e, se não houver nenhuma solução, a resposta do programa é *não*.

Caso contrário, uma das possíveis soluções é apresentada no ecrã, com a possibilidade de verificar outras possíveis soluções.

Caso o utilizador deseje ver mais resultados, basta recusar o atual, escrevendo "no" ou ";" na consola.

### 3.4 Estratégia de Pesquisa

Após a definição das variáveis e das restrições, foi usado o predicado *labeling* para calcular as possíveis soluções, com as opções padrão. O segundo argumento do predicado foi uma lista de listas simples, com as todas as restrições e variáveis definidas anteriormente.

## 4 Visualização da Solução

De modo a visualizar a solução encontrada, é utilizado o predicado *portray-clause(+Clause)*, como ilustrado no exemplo abaixo:

```
[0,1,0,1].
[3,0,0,1].
[1,1,1,2].
[0,3,1,0].
S = [[0,1,0,1],[3,0,0,1],[1,1,1,2],[0,3,1,0]] ?
```

**Figura 2.** Exemplo de uma solução

Cada **0** na matrix representa uma célula escurecida.

## 5 Resultados

De maneira a exemplificar algumas instâncias do problema, foram feitos testes para uma matrix 4\*4 e uma matriz 6\*6, utilizando os seguintes testes:

```
test1(Solution):-
    Left = [_ , [3, 1], [5], [4]],
    Up = [[4], [1, 4], [2], [4]],
    japaneseSum(Left, Up, Solution).

test2(Solution):-
    Left = [_ , [3, 1], [5], [4], [7 , 1], [2, 2]],
    Up = [[4], [1, 4], [2], [4], _ , [2, 6]],
    japaneseSum(Left, Up, Solution).
```

**Figura 3.** Teste com Matrix 4\*4 e com Matrix 6\*6

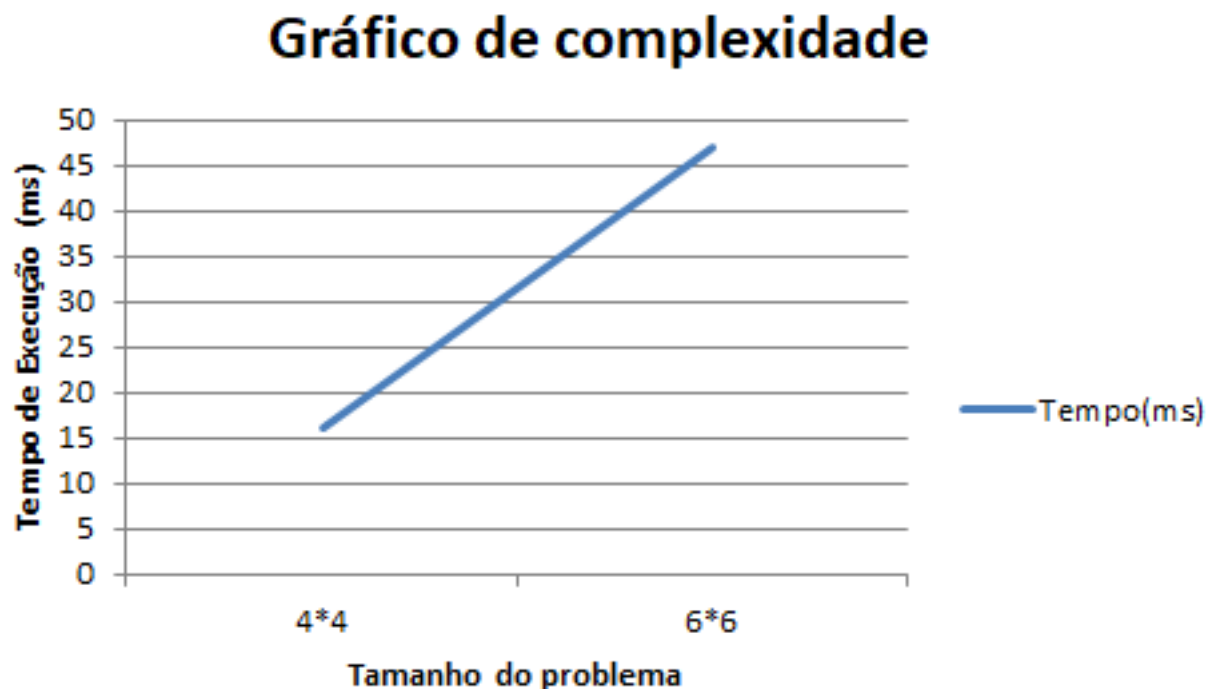
Com o uso do predicado *call time*, foi possível calcular o tempo de execução de cada teste, chamando na consola *call time(teste1(S), T*.

Este predicado utiliza o predicado *statistics* para obter o tempo.

```
% -----Used to calculate the execution time-----
call_time(G,T) :-
    statistics(runtime,[T0|_]),
    G,
    statistics(runtime,[T1|_]),
    T is T0 - T1,
    write(T0), nl,
    write(T1), nl,
    write(T).
```

**Figura 4.** T é o tempo de execução do predicado G

Após a obtenção dos tempos de execução, foi possível a realização dos gráficos seguintes:



Consegue-se que concluir que, o problema é de complexidade espacial quadrática, pois quanto maior a largura da matriz, o número de células aumenta de forma quadrática e é de complexidade temporal linear, pois é percorrida cada linha e cada coluna individualmente.

Apesar deste aspeto, o tempo de execução é bastante reduzido, o que permite uma rápida resolução do problema.

## 6 Conclusões

Após a utilização da biblioteca de PROLOG **CLP(FD)**, conclui-se que o uso de restrições é assaz eficiente e extremamente rápido para o desenvolvimento de várias aplicações, independentemente das ferramentas utilizadas. A vantagem da solução proposta é ser uma forma simples e eficiente de resolver o problema. No entanto está limitada por só poder preencher cada célula com um número entre

0 e 9, e não com um número retirado de um conjunto dado como foi proposto inicialmente.

## Referências

1. <https://maybepuzzles.wordpress.com/types/japanese-sums/>

## 7 Anexos

### japanese sums

```
1 :- use_module(library(clpfd)).
2 :- use_module(library(lists)).
3 :- use_module(library(statistics)).
4
5 % -----Returns the Solution with Left sums(lines) and Up
   ↪ sums(columns)-----
6 japaneseSum(Left, Up, Solution) :-
7     length(Left, Length),
8     length(Solution, Length),
9     maplist(same_length(Up), Solution),
10    append(Solution, FlatSolution),
11    domain(FlatSolution, 0, 9),
12    getLines(Solution, Left),
13    transpose(Solution, TransposedSolution),
14    getLines(TransposedSolution, Up),
15    transpose(TransposedSolution, FinalSolution),
16    labeling([], FlatSolution),
17    maplist(portray_clause, Solution).
18    %call_time(true,T_ms).
19
20
21 % -----Restrains each Line of the Board, using the getLine
   ↪ predicate-----
22 getLines([], []).
23 getLines([HLines | TLines], [HSums | TSums]) :-
24     getLine(HLines, HSums),
25     getLines(TLines, TSums).
26
27 % -----Generates a line and restrains each element to be between
   ↪ 0 and, at max, the maximum number of the Sums list and calls
   ↪ restrictBySums-----
28 getLine(_, []).
29 getLine(Line, Sums) :-
30     maximum(Max, Sums),
```

```

31     domain(Line, 0, Max),
32     restrictBySums(Line, Sums).
33
34 % -----Restrains each Line sum to be equal to the value(Number)
35   ↳ from the Sums list-----
36 restrictBySums(Line, [Number]) :-
37     sum(Line, #=, Number),
38     restrictConsecutive(Line).
39
40 restrictBySums(Line, Sums) :-
41     length(Sums, Length),
42     length(LineParts, Length),
43     append(LineParts, Line),
44     restrictMultipleSums(LineParts, Sums).
45
46 % -----Restrains the line so it has one division. Called if
47   ↳ the Sums List length is less than 2-----
48 restrictConsecutive(Line) :-
49     restrictConsecutiveAux(Line, 0, 0).
50
51 restrictConsecutiveAux([], _, NewCount) :- NewCount #=< 2.
52 restrictConsecutiveAux([H | T], Before, Count) :-
53     (Before #= 0 #/\ H #> 0) #<=> Begin,
54     (Before #> 0 #/\ H #= 0) #<=> End,
55     NewCount #= Count + Begin + End,
56     restrictConsecutiveAux(T, H, NewCount).
57
58 % -----Restrains the line if the Sums List length is bigger
59   ↳ than 1-----
60 restrictMultipleSums([HLine], [HSum]) :-
61     sum(HLine, #=, HSum),
62     restrictConsecutive(HLine),
63     length(HLine, Length),
64     Length #> 0.
65
66 restrictMultipleSums([HLine | TLine], [HSum | TSum]) :-
67     sum(HLine, #=, HSum),
68     restrictConsecutive(HLine),
69     length(HLine, Length),
70     Length #> 0,
71     element(Length, HLine, 0),
72     restrictMultipleSums(TLine, TSum).
73
74 % -----Used to calculate the execution time-----
75 call_time(G,T) :-

```

```

73     statistics(runtime,[T0|_]),
74     G,
75     statistics(runtime,[T1|_]),
76     T is T0 - T1,
77     write(T0), nl,
78     write(T1), nl,
79     write(T).
80
81
82 % -----An example test-----
83 test1(Solution):-
84     Left = [_ , [3, 1], [5], [4]],
85     Up = [[4], [1, 4], [2], [4]],
86     japaneseSum(Left, Up, Solution).
87
88 test2(Solution):-
89     Left = [_ , [3, 1], [5], [4], [7 , 1], [2, 2]],
90     Up = [[4], [1, 4], [2], [4], _ , [2, 6]],
91     japaneseSum(Left, Up, Solution).

```