Wrong Products

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

João Romão and Rafael Cristino

FEUP-PLOG, Turma 3MIEIC02, Grupo Wrong Products_2

Resumo Este trabalho foi desenvolvido com o intuito de dar resposta a um problema de satisfação de restrições. O problema em questão é o puzzle *Wrong Products*.
Foi utilizada a biblioteca *clpfd* do SICStus PROLOG de modo a resolver o problema, utilizando programação com restrições.
Foram obtidas soluções válidas para *inputs* de diversos tamanhos, sempre que apresentem soluções possíveis.
Foram ainda estudadas diversas formas de gerar inputs válidos para o problema em questão.

Keywords: Programação com Restrições · Prolog · SICStus

1 Introdução

O trabalho ao qual este documento se refere tem como intuito dar resposta a um problema, que consiste na resolução de um puzzle, modelando-o como um problema de satisfação de restrições.

O puzzle em questão é denominado *Wrong Products*, um puzzle que consiste na colocação de um conjunto de números numa grelha quadrada respeitando um certo conjunto de restrições.

O artigo terá então a seguinte estrutura:

- **Descrição do problema** - breve descrição do problema em questão.
- **Abordagem** - descrição das variáveis de decisão e das restrições aplicadas.
- **Visualização da Solução** - explicação sucinta dos predicados que permitem a visualização da solução encontrada.
- **Experiências e Resultados** - análise dimensional e diferentes estratégias de pesquisa.
- **Conclusões** - conclusões e trabalho futuro.
- **Referências**
- **Anexos** - resultados detalhados, como visualização de resultados e código utilizado.

2 Descrição do Problema

A resolução do puzzle *Wrong Products* é um problema de decisão que consiste em colocar numa grelha, onde as colunas e as linhas se encontram numeradas (ver a Fig. 1), números fornecidos (por exemplo, de 1 a 8) de tal forma que a multiplicação dos números que se encontram na mesma linha ou coluna seja superior ou inferior, em uma unidade, à numeração da mesma (ver a Fig. 1). Cada linha e cada coluna da grelha tem de conter, obrigatoriamente, dois números.

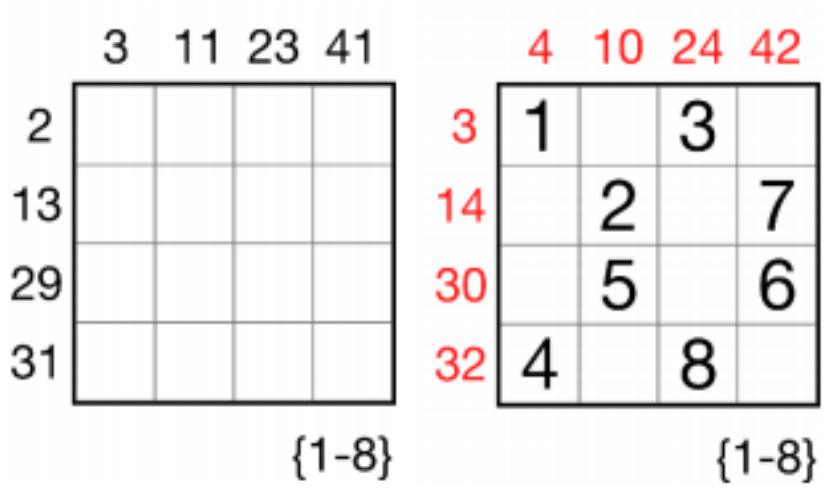


Figura 1. Grelha do puzzle, sem solução, à esquerda.
Grelha do puzzle, com solução e valores das multiplicações dos números (por linha e coluna), à direita.

3 Abordagem

3.1 Variáveis de decisão

Uma vez que a grelha fornecida como parâmetro pode tomar qualquer tamanho nxn, $\forall n, n > 1$, para dar resposta ao problema apenas interessa saber que posição ocupam os números indicados como parâmetro.

Para tal optou-se por representar estas variáveis numa lista, onde cada elemento é uma lista de dois valores: o primeiro o número da coluna do elemento e o segundo o valor do elemento.

Uma vez que cada linha e coluna apenas podem possuir dois elementos, partiu-se do princípio que a lista de elementos seria sequencial onde cada par de elementos corresponderia a uma linha do tabuleiro, não sendo deste modo necessária a representação do número da linha do elemento como variável de decisão. A forma de representação das variáveis de decisão pode ser observada nos anexos, secção 7.1.

Assim sendo, o domínio da variável que representa o número a ser colocado no tabuleiro é dado pelo primeiro argumento do predicado *solve* (uma lista de dois números, sendo o primeiro o menor número a colocar no tabuleiro e o segundo o maior número a colocar no tabuleiro).

O número da coluna é um valor $\in [1, N]$.

3.2 Restrições

Para o problema se considerar resolvido, as seguintes restrições têm de ser obrigatoriamente satisfeitas:

1. Cada um dos números colocados na grelha deve ser único e pertencer à lista de números fornecida.

   – Esta restrição é assegurada através do uso dos predicados *all_distinct/1* e *domain/3*, como pode ser visto nos anexos, secção 7.3.

2. Cada coluna, bem como cada linha, apenas pode ter duas células preenchidas.

   – Esta restrição é assegurada nas colunas através do uso do predicado *global_cardinality/1*, que garante que o número de cada coluna apenas apareça duas vezes na lista das variáveis de decisão (que foi descrita na secção 3.1), como pode ser visto nos anexos, secção 7.4.

3. A multiplicação dos valores dos dois elementos de cada linha ou coluna deve ser superior ou inferior, em uma unidade, ao número dessa linha ou coluna.

   – Esta restrição é assegurada através do uso dos predicados *checkRowMult(+RowList, +Vars)*, onde **RowList** é uma lista com os números de cada linha e **Vars** é a lista de variáveis de decisão, na ordenação original (por linha), e *checkColMult(+ColList, +VarsSorted)*, onde **ColList** é uma lista com os números de cada coluna e **VarsSorted** é a lista de variáveis de decisão, desta vez ordenadas por coluna. Para ordenar a lista de variáveis de decisão é utilizado o predicado *keysorting/2*. Este processo pode ser visto nos anexos, secção 7.5.

4. A cada célula da grelha pode apenas corresponder um elemento.

– Esta restrição é assegurada pelo uso do predicado *distinctColumnPosition(+PosList)* que certifica que as colunas dos dois elementos que se encontram na mesma linha são diferentes, como pode ser visto nos anexos, secção 7.6.

4 Visualização da Solução

O predicado principal que permite a visualização da solução é *solve(+LimitList, +ColsAndRows, -Res)*, onde o parâmetro **LimitList** representa o domínio das variáveis numéricas a colocar no tabuleiro e **ColsAndRows** uma lista com dois elementos: o primeiro uma lista contendo a numeração sequencial de cada coluna e o segundo a numeração sequencial de cada linha.

Este predicado é também responsável por chamar o predicado *display_solution/3* que permite uma melhor visualização dos resultados no formato de um tabuleiro, fazendo a interpretação dos valores atribuídos às variáveis de decisão representativas do valor de cada peça. Podem ser consultados nos anexos, secção 7.2, exemplos do *output* gráfico (texto) da resolução do problema.

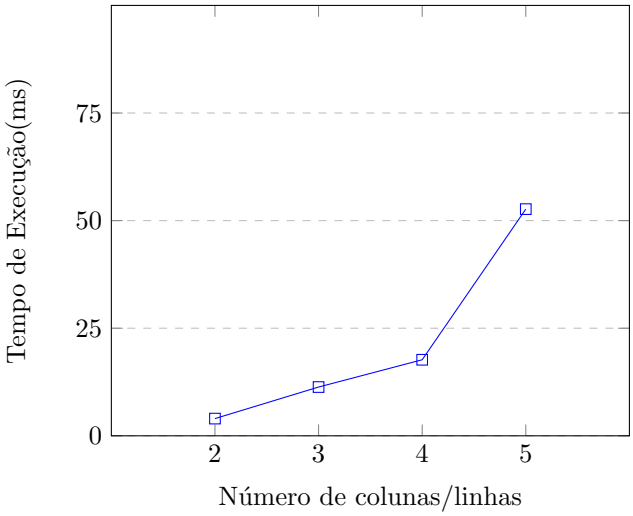
5 Experiências e Resultados

5.1 Análise Dimensional

O tabuleiro representativo pode apresentar diversos tamanhos quadrados, fazendo variar a dimensão dos dados a analisar, tendo sido encontradas soluções para todas as situações testadas.

Tabela 1. Exemplos de visualização em texto de resultados para problemas de diferentes dimensões tables.

Solução 3x3	Solução 4x4	Solução 5x5
5 17 11 5 _ 3 2 25 4 6 _ 6 1 _ 5	34 3 23 11 6 5 1 _ _ 7 _ 4 _ 2 17 _ _ 3 6 55 7 _ 8 _	29 81 7 10 27 11 5 _ 2 _ _ 29 _ 10 3 _ _ 73 _ 8 _ 9 _ 8 _ _ _ 1 7 23 6 _ _ _ 4



Os dados utilizados no gráfico podem ser observados também na tabela 2, secção 7.7.

5.2 Estratégias de Pesquisa

De modo a gerar dinamicamente os problemas a resolver, foram abordadas diferentes estratégias.

Inicialmente optou-se por uma estratégia aleatória, gerando o cabeçalho de cada coluna e linha aleatoriamente, limitados por valores possíveis (predicado *generateRandom(+BoardSize, -Problem)*). O programa gera continuamente novos resultados até ser encontrado um problema que seja possível resolver.

Após uma análise dos dados de entrada válidos, chegou-se à conclusão de que a multiplicação dos cabeçalhos das colunas e das linhas (não os dados de entrada mas o resultado da multiplicação de cada elemento presente numa linha ou coluna, representado a vermelho na figura 1), seriam iguais e teriam um valor constante associado à dimensão do tabuleiro.

Por exemplo, no caso da figura 1, o resultado da multiplicação dos cabeçalhos das colunas seria:

$$4 * 10 * 24 * 42 = 40320 \tag{1}$$

e o da multiplicação dos cabeçalhos das linhas seria:

$$3 * 14 * 30 * 32 = 40320 \tag{2}$$

O valor que a multiplicação da lista de cabeçalhos das colunas e das linhas toma quando se trata de um problema válido é a seguinte:

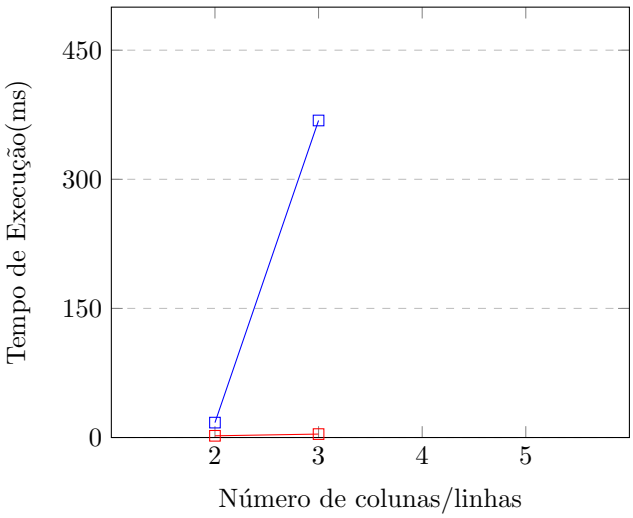
$$x_2 = 24 \tag{3}$$

$$x_n = x_{n-1} * 2 * n * (2 * n - 1) \tag{4}$$

sendo n o valor correspondente ao número de colunas/linhas do tabuleiro.

Apesar desta última restrição, implementada no predicado *generateHeuristic(+BoardSize, -Problem)*, ter melhorado os tempos de pesquisa por um problema válido e ser uma condição necessária para os dados de entrada constituírem uma solução válida, não é uma condição suficiente, não sendo assim suficiente para gerar problemas com grelhas de grandes dimensões eficientemente.

De seguida encontra-se a visualização dos tempos de execução de cada estratégia de geração de problema (azul: estratégia aleatória, vermelho: estratégia baseada na heurística apresentada).



Os dados utilizados no gráfico podem ser observados também nas tabelas 3 (aleatória) e 4 (heurística), secção 7.7. Para tamanhos de grelha superiores a três, com os métodos desenvolvidos, não é possível gerar problemas eficientemente, como descrito acima.

6 Conclusões e Trabalho Futuro

A realização deste trabalho permitiu um primeiro contacto com programação por restrições, assim como os benefícios que esta traz. Permitiu a obtenção de resultados de uma forma mais intuitiva e eficiente através da aplicação de restrições.

Os resultados obtidos, para diferentes tamanhos de tabuleiro e diferentes quantidades de "peças"(números), estão de acordo com o pretendido, apresentando-se como soluções válidas para o problema em causa.

Não foram encontradas limitações no método utilizado, a não ser a eventual possibilidade de melhoria de performance.

Quanto à geração dinâmica de problemas, o enorme número de possibilidades que cada campo pode tomar, foi um obstáculo encontrado, quando os dados do problema são de elevada dimensão, contudo mostrou-se bem sucedida para dados de dimensão reduzida.

Deste modo este seria o principal aspecto a melhorar, algo que requereria uma investigação profunda e análise dos possíveis padrões que um problema válido pode tomar.

Referências

1. We Are Puzzlers Club: Part 2, https://logicmastersindia.com/lmitests/dl.asp?attachmentid=790view=1. Acedido pela última vez em 4 de janeiro de 2021
2. SICStus Prolog Documentation, https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/. Acedido pela última vez em 4 de janeiro de 2021
3. Silva D.: Programação em Lógica com Restrições (2020)
4. Silva D.: Programação em Lógica com Restrições no SICStus Prolog (2020)

7 Anexos

7.1 Variáveis de Decisão

```
solve([LowerLimit, UpperLimit], [Columns, Rows], Res) :-
    generateVars(Columns, Rows, Vars),
    generateNumPosList(Vars, NumList, PosList),
    (...)

makeLength2(X) :-
    length(X, 2).

generateVars(Columns, Rows, Vars) :-
    length(Columns, ColumnsLength),
    length(Rows, RowsLength),
    ColumnsLength == RowsLength,
    Size is RowsLength * 2,
    length(Vars, Size),
    maplist(makeLength2, Vars).

generateNumPosList([], [], []).
generateNumPosList([[Pos, Num] | VarsT], [NumListH | NumListT],
  [PosListH | PosListT]) :-
    NumListH = Num,
    PosListH = Pos,
    generateNumPosList(VarsT, NumListT, PosListT).
```

7.2 Exemplos de Execução

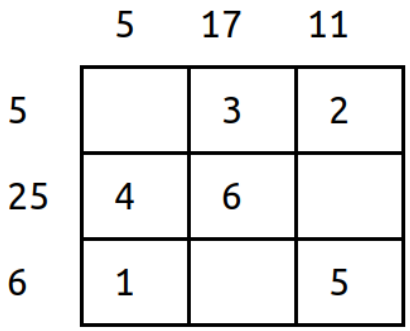


Figura 2. *Output* da resolução de um problema com grelha de dimensão 3x3.

	34	3	23	11
6	5	1		
7		4		2
17			3	6
55	7		8	

Figura 3. *Output* da resolução de um problema com grelha de dimensão 4x4.

	29	81	7	10	27
11	5		2		
29		10	3		
73		8		9	
8				1	7
23	6				4

Figura 4. *Output* da resolução de um problema com grelha de dimensão 5x5.

7.3 Restrição 1

Cada um dos números colocados na grelha deve ser único e pertencer à lista de números fornecida.

```
solve([LowerLimit, UpperLimit], [Columns, Rows], Res) :-
    (...)

    domain(NumList, LowerLimit, UpperLimit),
    length(Columns, ColNum),
    domain(PosList, 1, ColNum),

    all_distinct(NumList),

    (...)
```

7.4 Restrição 2

Cada coluna, bem como cada linha, apenas pode ter duas células preenchidas.

```
solve([LowerLimit, UpperLimit], [Columns, Rows], Res) :-
    (...)

    generateCardinalityList(ColNum, CardinalityList),
    global_cardinality(PosList, CardinalityList),

    (...)

generateCardinalityList(0, []).
generateCardinalityList(N, [ H | T ]) :-
    H = N-2,
    N1 is N - 1,
    generateCardinalityList(N1, T), !.
```

7.5 Restrição 3

A multiplicação dos valores dos dois elementos de cada linha ou coluna deve ser superior ou inferior, em uma unidade, ao número dessa linha ou coluna.

```
solve([LowerLimit, UpperLimit], [Columns, Rows], Res) :-
    (...)

    checkRowMult(Rows, Vars),

    length(Vars, _Len),
    length(VarsSorted, _Len),
    maplist(makeLength2, VarsSorted),

    keysorting(Vars, VarsSorted),
```

```
    checkColMult(Columns, VarsSorted),

    (...)

makeLength2(X) :-
    length(X, 2).

checkColMult([], []).
checkColMult([ CurrentCol | Columns ], [ [_, Value1],
  [_, Value2] | VarsSortedT ]) :-
    abs(Value1 * Value2 - CurrentCol) #= 1,
    checkColMult(Columns, VarsSortedT).

checkRowMult([], []).
checkRowMult([ CurrenRow | Rows ], [ [_, Value1], [_, Value2] | VarsT ]) :-
    abs(Value1 * Value2 - CurrenRow) #= 1,
    checkRowMult(Rows, VarsT).
```

7.6 Restrição 4

A cada célula da grelha pode apenas corresponder um elemento.

```
solve([LowerLimit, UpperLimit], [Columns, Rows], Res) :-
    (...)

    distinctColumnPosition(PosList),

    (...)

distinctColumnPosition([]).
distinctColumnPosition([Pos1, Pos2 | T]) :-
    Pos1 #\= Pos2,
    distinctColumnPosition(T).
```

7.7 Tabelas de Desempenho

Tabela 2. Tempos de execução da resolução do problema para diferentes dimensões

Dimensão do problema (n)	Tempo de execução (ms)
2	$\frac{3+5+4}{3} = 4$
3	$\frac{10+11+13}{3} = 11.3(3)$
4	$\frac{20+17+16}{3} = 17.6(6)$
5	$\frac{49+50+59}{3} = 52.6(6)$

Tabela 3. Tempos de execução da geração do problema (método aleatório) para diferentes dimensões

Dimensão do problema (n)	Tempo de execução (ms)
2	$\frac{13+13+26}{3} = 17.3(3)$
3	$\frac{351+309+445}{3} = 368.3(3)$

Tabela 4. Tempos de execução da geração do problema (método baseado na heurística implementada) para diferentes dimensões

Dimensão do problema (n)	Tempo de execução (ms)
2	$\frac{3+2+1}{3} = 2$
3	$\frac{5+4+3}{3} = 4$