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[1].

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 respetiva 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.

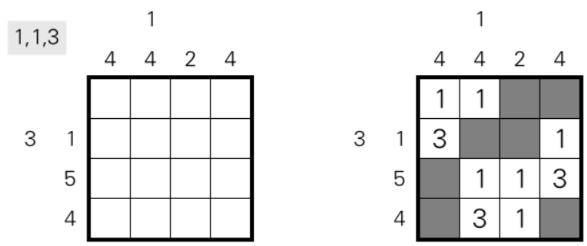


Figura 1. Exemplo de um tabuleiro preenchido

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

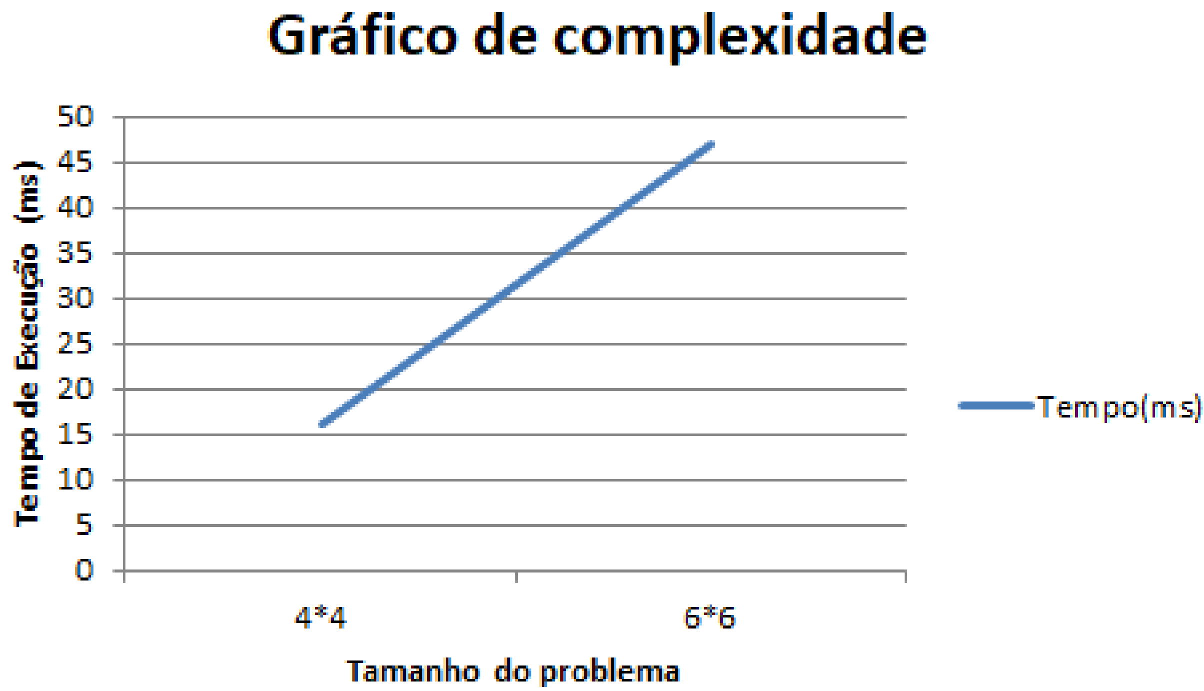
Com o uso do predicado *call time*, foi possível calcular o tempo de execução de cada teste, chamando na consola *call time(teste1(S), T*.

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 obtençao 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 a de 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

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

% ----------Returns the Solution with Left sums(lines) and Up sums(columns)--------------
japaneseSum(Left, Up, Solution) :-
  length(Left, Length),
  length(Solution, Length),
  maplist(same_length(Up), Solution),
  append(Solution, FlatSolution),
  domain(FlatSolution, 0, 9),
  getLines(Solution, Left),
  transpose(Solution, TransposedSolution),
  getLines(TransposedSolution, Up),
  transpose(TransposedSolution, FinalSolution),
  labeling([], FlatSolution),
  maplist(portray_clause, Solution).
  %call_time(true,T_ms).


% ----------Restrains each Line of the Board, using the getLine predicate--------------
getLines([], []).
getLines([HLines | TLines], [HSums | TSums]) :-
  getLine(HLines, HSums),
  getLines(TLines, TSums).

% ------Generates a line and restrains each element to be between 0 and, at max, the maximum number of the Sums list and calls restrictBySums--------------
getLine(_, []).
getLine(Line, Sums) :-
  maximum(Max, Sums),
```

```
  domain(Line, 0, Max),
  restrictBySums(Line, Sums).

% -------Restrains each Line sum to be equal to the value(Number) from the Sums list--------------
restrictBySums(Line, [Number]) :-
  sum(Line, #=, Number),
  restrictConsecutive(Line).

restrictBySums(Line, Sums) :-
  length(Sums, Length),
  length(LineParts, Length),
  append(LineParts, Line),
  restrictMultipleSums(LineParts, Sums).

% ----------Restrains the line so it has one division. Called if the Sums List length is less than 2--------------
restrictConsecutive(Line) :-
  restrictConsecutiveAux(Line, 0, 0).

restrictConsecutiveAux([], _, NewCount) :- NewCount #=< 2.
restrictConsecutiveAux([H | T], Before, Count) :-
  (Before #= 0 #/\ H #> 0) #<=> Begin,
  (Before #> 0 #/\ H #= 0) #<=> End,
  NewCount #= Count + Begin + End,
  restrictConsecutiveAux(T, H, NewCount).

% ----------Restrains the line if the Sums List length is bigger than 1--------------
restrictMultipleSums([HLine], [HSum]) :-
  sum(HLine, #=, HSum),
  restrictConsecutive(HLine),
  length(HLine, Length),
  Length #> 0.

restrictMultipleSums([HLine | TLine], [HSum | TSum]) :-
  sum(HLine, #=, HSum),
  restrictConsecutive(HLine),
  length(HLine, Length),
  Length #> 0,
  element(Length, HLine, 0),
  restrictMultipleSums(TLine, TSum).

% ----------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).


% ----------An example test--------------
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).
```