Doppelblock:
Uma abordagem em PLR

António Almeida - up201505836
João Damas - up201504088

FEUP-PLOG, Turma 3MIEIC1, Grupo Doppelblock_5

Resumo Este trabalho foi desenvolvido no âmbito da Unidade Curricular PLOG e consiste numa abordagem ao puzzle Doppelblock segundo uma perspetiva de programação em lógica por restrições. Foram desenvolvidos algoritmos para resolver e gerar instâncias do puzzle, assim como feita uma análise estatística (da eficiência) e crítica (do desenvolvimento) relativamente ao resultado obtido, concluindo que esta abordagem permite a conceção de algoritmos curtos e relativamente eficientes para problemas deste tipo.

Keywords: doppelblock, puzzle, plog, plr, prolog

1 Introdução

Este trabalho, realizado no âmbito da UC PLOG, tem como principal motivação a consolidação de conhecimentos adquiridos relativamente a PLR. Para tal, optou-se por explorar um problema de decisão combinatória, nomeadamente o puzzle Doppelblock. Como principal objetivo assinala-se a construção de algoritmos, em PLR, para resolver e gerar instâncias do mesmo.

As secções seguintes descrevem, de uma forma mais pormenorizada, o problema em questão, a abordagem tomada, nomeadamente ao nível de variáveis de decisão, restrições e estratégias de pesquisa, os resultados de um pequeno estudo estatístico sobre instâncias de diferentes complexidades do problema e algumas conclusões sobre o trabalho desenvolvido e possíveis melhorias.

2 Descrição do Problema

Doppelblock é um puzzle realizado num tabuleiro quadrado de dimensão variável. O objetivo é preencher o tabuleiro, sendo que uma determinada posição poderá ficar a preto ou ocupada por um número de 1 a N-2, sendo N a dimensão do tabuleiro. Em cada linha/coluna do tabuleiro devem ser usados todos os números de 1 a N-2 e devem existir exatamente duas posições a preto.

Como input inicial, é dado a conhecer o valor das somas objetivo para as linhas/colunas do tabuleiro (eventualmente algumas poderão ser omitidas, aumentando a dificuldade sem aumentar o tamanho do tabuleiro). O valor para uma dada linha/coluna corresponde à soma dos números colocados entre as duas

posições preenchidas a preto na mesma. Nas figuras 1 e 2 podem observar-se um exemplo de uma instância que ilustra as regras supradescritas.

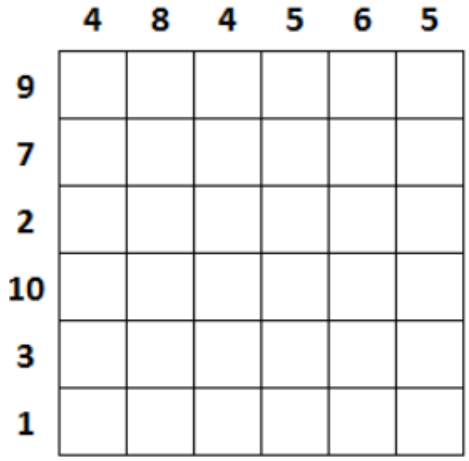

Figura 1. Exemplo de instância inicial do puzzle com as somas nas respetivas linhas/colunas nas bordas.

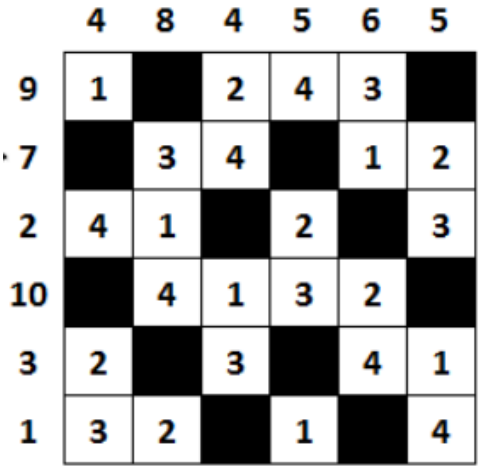

Figura 2. Possível solução para a instância inicial apresentada.

3 Abordagem

Nas subsecções seguintes será necessário recorrer frequentemente ao tamanho do tabuleiro numa instância genérica do puzzle. Assim, e até ao fim da secção, N representa o tamanho do tabuleiro numa instância válida do puzzle.

3.1 Variáveis de Decisão

Visto que o objetivo é preencher o tabuleiro, pode-se dizer que existem N*N variáveis de domínio. Tendo sido convencionado que uma posição a preto é internamente representada por 0, o domínio das variáveis de decisão é [0, N-2]. Abaixo encontra-se o excerto de código que trata de criar e assegurar o domínio das variáveis de decisão.

```
solveInstance(LineSums, ColumnSumns):-
...
    % Get the NxN board domain variables
    getInitialBoard(N, Board),

    % Ensure domain of every value on the board (is in range [0,N-2])
    % Note that 0 = black cell
    ensureDomain(Limit, Board),
...

% Ensures the domain of the variables used
ensureDomain(_, []).
ensureDomain(Limit, [Line | Remaining]):-
    domain(Line, 0, Limit),
    ensureDomain(Limit, Remaining).
```

3.2 Restrições

Existem duas grandes restrições que definem o problema:

Em cada linha/coluna os valores devem ser distintos, existindo duas posições a preto

Esta restrição poderia ser dividida em duas distintas, no entanto, sendo implementadas na mesma altura, resolveu-se juntá-las. A implementação faz uso do predicado global_cardinality/2, sendo a lista de pares Value-Amount obtida dinamicamente consoante o valor de N.

```
getValueAmountPairList(0, [0-2]). % Each value comes up once, except for 0 (black cell)
getValueAmountPairList(N, PairList):-
    N > 0,
    N1 is N-1,
    getValueAmountPairList(N1, PairListTemp),
    append(PairListTemp, [N-1], PairList). %Value should show up exactly once
```

Esta lista é depois aplicada a todas as linhas e todas as colunas através do predicado maplist/2.

```
solveInstance(LineSums, ColumnSums):-
...
% Restriction 1
    % Ensure that numbers in each row/column are distinct
    % Also ensures exactly two black cells per row/column
    % Note that Limit = N-2
    ensureAllDistinct(Limit, Board),
...

% Ensures that all elements in a line/column are different
% Exception are black cells, which there are 2 of
ensureAllDistinct(Limit, Board):-
    getValueAmountPairList(Limit, PairList),
    maplist(allDistinct(PairList), Board),
    transpose(Board, TransposedBoard),
    maplist(allDistinct(PairList), TransposedBoard).

allDistinct(PairList, Line):-
    global_cardinality(Line, PairList).
```

A soma dos elementos entre duas posições a preto numa linha/coluna deve ser um valor especificado

Para esta restrição foi utilizado um autómato finito determinista que permite garantir os momentos em que a soma de elementos deve ser efetuada e guardar a soma acumulada. Na figura 3 encontra-se o DFA que foi implementado.

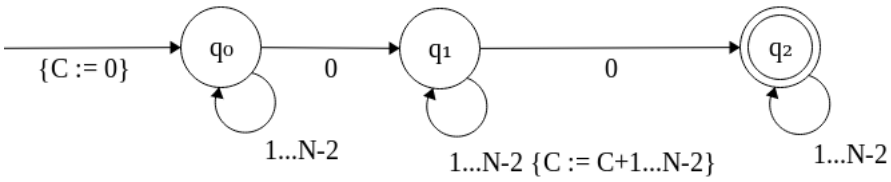

Figura 3. DFA representado a segunda restrição. Inicialmente os valores são ignorados (para efeitos da soma). Encontrando o primeiro 0, a soma comea a acumular com os elementos seguintes, ficando concluída quando for encontrado o segundo 0, altura em que os elementos voltam a ser, na prática, ignorados.

Em SICStus Prolog, a restrição foi implementada pelo predicado automaton/8. A lista de transições é obtida dinamicamente em função de N. Para cada valor entre 1 e N-2 existem três transições a assinalar: no estado inicial, q0, o

número não afeta a soma, ocorrendo a passagem ao próximo elemento; no estado intermédio, q1, é somado o valor do elemento à soma acumulada no contador; no estado final, q2, o valor volta a não interferir na soma, que nesta altura já se encontra calculada.

```
solveInstance(LineSums, ColumnSums):-
...
% Restriction 2
    % Ensures sum between black cells in each line/column is specified value (if specified)
    maplist(ensureSums(Limit), Board, LineSums),
    transpose(Board, Transposed),
    maplist(ensureSums(Limit), Transposed, ColumnSums),
...

% Base case, add transitions that ensure automaton progression
getTransitions(0, AuxList, TransitionList, _):-
    append(AuxList, [arc(q0,0,q1), arc(q1,0,q2)], TransitionList).
% Recursive case, add transitions in 'waiting' states
% and intermediate state transition where the counter is updated
getTransitions(N, AuxList, TransitionList, Counter):-
    N > 0,
    append(AuxList, [arc(q0,N,q0), arc(q1,N,q1,[Counter+N]), arc(q2,N,q2)], NewAuxList),
    N1 is N-1,
    getTransitions(N1, NewAuxList, TransitionList, Counter).

ensureSums(Limit, _, -1). %If sum has not been specified
ensureSums(Limit, Line, Sum):-
    getTransitions(Limit, [], TransitionList, C),
    automaton(Line, _, Line, [source(q0), sink(q2)], TransitionList, [C], [0], [Sum]).
```

De notar que, apesar da lista de transições ser sempre a mesma, esta é recalculada a cada chamada ao predicado ensureSums. Isto deve-se ao facto de nas transições geradas dinamicamente ser necessário utilizar uma variável como contador e da consequente necessidade de utilizar a mesma variável na criação dos factos arc e na chamada a automaton.

3.3 Estratégia de Pesquisa

Como estratégia de pesquisa foram utilizadas as opções bisect e down no labeling. Enquanto que a primeira garante uma mais rápida convergência para o valor correto, a segunda garante que os valores são testados a partir do limite superior do domínio. Desta forma, o valor, não sendo correto, terá maior probabilidade de ultrapassar o valor da soma e falhar rapidamente, permitindo uma rápida passagem ao próximo. Este conjunto de opções revelou-se vantajosa, visto que, na generalidade, reduziu o tempo de execução e/ou o número de backtracks efetuados, como observável na tabela 1.

Tabela 1. Estatísticas de execução com e sem as opções de labeling

Puzzle instance		**Stats w/o labeling options**		**Stats w/ labeling options**	
Line Sums	*Column Sums*	*Time*	*#Backtracks*	*Time*	*#Backtracks*
9,7,2,10,3,1	4,8,4,5,6,5	0.01	233	0.01	139
4,5,4,4,8,4,0	10,4,4,4,5,5,12	10.4	106317	0.02	138
11,5,15,5,12,0,2	10,10,9,10,4,10,0	1.66	17245	1.12	9651

3.4 Gerador dinâmico de instâncias

Para tornar a tarefa de gerar instâncias do puzzle a resolver mais acessível, foi desenvolvido um simples gerador. A abordagem é semelhante à do algoritmo que resolve instâncias, sendo a principal diferença a substituição da segunda restrição por um conjunto de restrições geradoras: assegurar que nenhuma soma assume o valor 0 (para o algoritmo não ser ganancioso e gerar o maior número de 0 que conseguir, tornando o puzzle demasiado trivial), assegurar que a soma na primeira linha é um valor aleatório na gama [0,Max], em que Max corresponde ao valor máximo possível, a soma dos números naturais até N-2, e assegurar que a soma das somas atinge um valor mínimo consoante a dificuldade pretendida pelo utilizador: valores mais altos são, geralmente, sinal de puzzles mais difíceis pelo aumento de possibilidades das combinações para obter o valor da soma.

```
generateInstance(N, LineSums, ColumnSums, Difficulty):-
    ... % Identical to solver

    % Ensure that no black cells are adjacent to avoid trivial cases
    ensureNoAdjacentBlackCells(Limit, Board),

    % Force line and column sums to be greater than a number, based on difficulty
    MaxSum is round(Limit*(Limit+1)/2), %round to avoid floating point numbers
    MaxSumLine is MaxSum*N,
    getMinSum(Difficulty, MaxSumLine, MinSum),
    sum(LineSums, #>, MinSum),
    sum(ColumnSums, #>, MinSum),

    % Force first sum to be a random value so it generates different boards
    now(Time),
    setrand(Time),
    random(1, MaxSum, FirstSum),
    Board = [H | T],
    ensureSums(Limit, H, FirstSum),

    ... % Identical to solver
```

4 Visualização da Solução

Para visualizar as soluções obtidas, foi desenvolvido um predicado que reproduz o tabuleiro do puzzle preenchido com as suas restrições e a solução respetiva.

```
getDisplay(0, '#'). % black cell
getDisplay(A, A).    % other digits

% Displays a board of side N with LineSums and ColumnSums as restrictions
displayBoard(Board, N, LineSums, ColumnSums):-
    nl, write('   '),
    displayBoardHeader(ColumnSums),
    displayBoardTail(Board, N, LineSums),
    write('  '),
    displaySeparator(N).

displayBoardTail([], _, _).
displayBoardTail([Line | RestLines], N, [Sum | RestSums]):-
    write('  '),
    displaySeparator(N),
    write('  '),
    displayLine(Line, Sum), nl,
    displayBoardTail(RestLines, N, RestSums).

displayLine([], Sum):- write('| - '), write(Sum).
displayLine([Value | T], Sum):-
    write('| '),
    getDisplay(Value, Display),
    write(Display), write(' '),
    displayLine(T, Sum).

displayBoardHeader([]):- nl.
displayBoardHeader([H|T]):-
    write(' '), write(H), write('  '),
    displayBoardHeader(T).

displaySeparator(0):- nl.
displaySeparator(N):-
    write('----'),
    N1 is N-1,
displaySeparator(N1).
```

```
  4   8   4   5   6   5
-----------------------
| 1 | # | 2 | 4 | 3 | # | - 9
-----------------------
| # | 3 | 4 | # | 1 | 2 | - 7
-----------------------
| 4 | 1 | # | 2 | # | 3 | - 2
-----------------------
| # | 4 | 1 | 3 | 2 | # | - 10
-----------------------
| 2 | # | 3 | # | 4 | 1 | - 3
-----------------------
| 3 | 2 | # | 1 | # | 4 | - 1
-----------------------
```

Figura 4. Exemplo de visualização de instância do puzzle resolvida, com as somas nas respetivas linhas/colunas nas bordas. As células pretas são representadas pelo carater '#'.

5 Resultados

Para testar a solução desenvolvida, foram observadas as estatísticas de execução de instâncias do puzzle de diferentes tamanhos, nomeadamente 5,6,7 e 8. Os resultados obtidos encontram-se na tabela 2.

Tabela 2. Estatísticas de execução para algumas instâncias, com e sem todas as somas fornecidas ao *solver*. Nas colunas Line Sums/Column Sums, valores sublinhados representam valores omitidos ao programa no caso em que apenas são fornecidas algumas somas, tal como nos puzzles originais, se for o caso.

			All sums provided		**Some sums omitted**	
Board Size	**Line Sums**	**Column Sums**	**Time**	**#Backtracks**	**Time**	**#Backtracks**
5	0,6,3,2,0	5,1,4,3,1	0	1	0.01	10
5	2,6,0,3,4	3,1,0,6,3	0	5	0.01	18
6	9,7,2,10,3,1	4,8,4,5,6,5	0.01	139	*	*
6	0,0,0,0,1,3	3,3,5,9,4,4	0	3	0.01	130
6	3,10,6,8,2,4	0,1,6,3,9,1	0.04	340	0.69	9798
6	7,3,0,0,0,1	3,8,3,3,6,4	0.01	65	0.03	395
7	4,5,4,4,8,4,0	10,4,4,4,5,5,12	0.02	138	9.48	138504
7	10,0,14,12,5,6,9	5,4,15,10,5,4,1	2.41	20669	126.11	1396555
7	11,5,15,5,12,0,2	10,10,9,10,4,10,0	1.12	9651	36.51	3422295
7	2,7,0,12,5,5,8	6,0,12,4,12,6,13	2.29	19150	25.25	303097
8	14,10,1,8,21,5,10,12	8,16,12,6,0,0,8,10	390.86	3487340	*	*

* - No puzzle original todos os valores eram fornecidos

Pelos resultados obtidos, é possível concluir que, para tabuleiros até tamanho 7, o algoritmo desenvolvido apresenta uma eficiência bastante satisfatória, com o tempo de execução, dadas todas as somas, sempre inferior a 2.5s. Uma outra conclusão é que o tempo de execução parece crescer (aproximadamente) de forma linear com o tamanho do tabuleiro. O número de backtracks também aparenta variar de forma análoga. Apesar disso, parece existir um terceiro fator que influencia o desempenho: o valor das somas. O principal destaque pretende-se com o segundo puzzle 6x6. O valor das somas era, em média, bastante baixo, o que permitiu uma mais rápida execução. Para os valores obtidos no caso em que apenas algumas somas são fornecidas, existe um aumento comparativamente com os mesmos dados para a instância com todas as somas dadas no mesmo puzzle, o que será normal, visto que tendo menos informação o problema estará menos restringido e será necessário voltar atrás mais frequentemente.

A partir de 8x8 a eficiência começa a descrescer de forma acentuada, como observável no aumento do tempo de execução e número de backtracks num fator de aproximadamente 170 e 182, respetivamente.

6 Conclusões e Trabalho Futuro

Com este trabalho, é possível concluir que a programação por restrições é uma ferramenta bastante útil, permitindo resolver diferentes tipos de problemas (no caso, um de decisão combinatória), mais uma vez através de algoritmos pequenos e simples. A grande variedade de predicados já incluídos na biblioteca CLPFD contribui, sem dúvida, para este facto.

Relativamente ao trabalho desenvolvido, a opinião final é positiva. A solução desenvolvida, com recurso a DFAs, revelou-se simples e prática. Um simples gerador também foi desenvolvido, permitindo a criação dinâmica do problema a ser resolvido, num tamanho e dificuldade definidos pelo utilizador, permitindo grande flexibilidade. Apesar disso, pelos resultados estatísticos obtidos, infere-se que algumas melhorias poderiam ser introduzidas, nomeadamente ao nível da eficiência do algoritmo de resolução para tabuleiros de dimensões mais elevadas.

Referências

1. Doppelblock Set of Rules, `http://logicmastersindia.com/lmitests/dl.asp?attachmentid=659&view=1`
2. SICStus Prolog User's Manual, `https://sicstus.sics.se/sicstus/docs/latest4/pdf/sicstus.pdf`

Anexos

Anexo I: Instruções de utilização

Para correr o programa, os seguintes passos devem ser executados, sequencialmente:

1. Consultar o ficheiro 'main.pl'
2. Correr o predicado doppelblock/0

O predicado doppelblock/0 mostrará um menu principal no qual as opções disponíveis são mostradas ao utilizador. Instruções de utilização para uma opção em particular também são mostradas ao escolher a mesma.