Doppelblock

Diogo Luís Rey Torres[1] e Francisco Teixeira Lopes[1]

[1] FEUP-PLOG, Turma 3MIEIC01, Grupo Doppelblock_3

Resumo. Este trabalho incide sobre a resolução de um jogo denominado DoppelBlock. O jogo consiste em preencher um tabuleiro com números e células pretas, com o objetivo das somas entre as células pretas em cada linha/coluna corresponderem ao valor especificado. Para resolver este problema, utilizou-se programação lógica com restrições com a biblioteca clpfd do Prolog. Após alguns testes de performance da solução desenvolvida, concluiu-se que o tempo de execução é exponencial para tamanhos do problema crescentes, e que, para tabuleiros diferentes mas do mesmo tamanho, as opções de labeling correspondem a tempos de execução que não variam linearmente, ou seja, em alguns casos uma opção é significativamente melhor, mas noutros, não.

1 Introdução

O trabalho foi desenvolvido no âmbito da unidade curricular de Programação Lógica, sendo que, o objetivo do trabalho foi criar um gerador de soluções para o jogo DoppelBlock, assim como, um gerador de problemas para o mesmo jogo.

Para gerar soluções deste jogo, e também problemas, recorreu-se a programação lógica com restrições, utilizando a biblioteca clpfd de Prolog.

O artigo divide-se em várias seções, começando pela descrição do problema, seguindo-se da explicação da abordagem ao problema de solução e geração de tabuleiros DoppelBlock, depois é apresentada uma secção sobre a visualização da solução e, finalmente, uma secção de avaliação de resultados obtidos.

2 Descrição do Problema

O jogo DoppelBlock consiste num tabuleiro quadrado, de dimensões variadas, em que para cada linha/coluna a soma dos valores compreendidos entre duas células pretas tem de corresponder a um determinado valor. O objetivo do jogo é, portanto, colocar os valores nas células, verticalmente ou horizontalmente, de modo que nessas linhas/colunas os valores não sejam repetidos (exceto as duas células pretas).

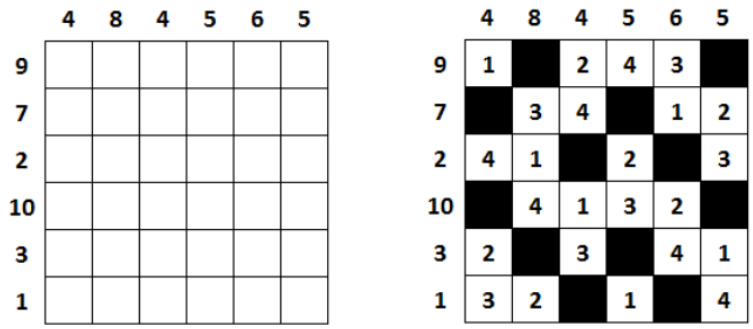


Figura 1 - exemplo de tabuleiro DoppelBlock

3 Abordagem (Resolução de problemas)

O tabuleiro de jogo, em Prolog, é representado segundo uma estrutura que contém toda a informação pertinente para a sua resolução. Então, esta contém: uma lista com 3 listas. A primeira lista, de tamanho N, contém os valores das somas respetivos para cada coluna, a segunda lista, também de tamanho N, contém os valores das somas para cada linha. E por fim, uma lista de listas de tamanho NxN que possui os valores de cada posição do tabuleiro.

3.1 Variáveis de Decisão

Dada a representação utilizada para o tabuleiro, o objetivo do solver será retornar uma lista com os valores de cada posição do tabuleiro de forma a respeitar as regras do jogo anteriormente referidas.

Relativamente ao tabuleiro, o seu tamanho depende do tamanho da lista que contém as somas das linhas/colunas. Visto que que o tabuleiro é quadrado, é gerado uma lista de listas com tamanho NxN em que o seu domínio para cada linha/coluna do tabuleiro encontra-se entre 0 e N-2.

```
% Create a NxN matrix
getInitialBoard(N, Board) :-
        length(Board, N),
        maplist(same_length(Board), Board).
```

Figura 2 - criação tabuleiro quadrado

```
% Define domain of each variable to values between 0 and N-2
defineDomain(N, Rows):-
        MaxN #= N - 2,
        maplist(define_domain(MaxN), Rows).

define_domain(MaxN,X):- domain(X, 0, MaxN).
```

Figura 3 - predicado de definição de domínio

3.2 Restrições

A resolução deste problema de decisão, na nossa perspetiva, divide-se exatamente em duas restrições:

- Cada linha/coluna só pode conter duas células pretas e cada número entre 1 e N-2 só pode aparecer exatamente uma vez
- Os valores somados entre duas células pretas em cada linha/coluna tem de respeitar um determinado valor

Para aplicar a primeira restrição, é definida a cardinalidade para cada linha e coluna através da função global_cardinality que recebe uma lista que obriga cada linha/coluna a conter exatamente 2 células pretas (representadas por 0) e números entre 1 e N-2 que só podem aparecer exatamente uma vez.

```
% For each row and each column put exactly two cells black and each number between 1 and N-2 appears exactly once
defineCardinality(N, Rows) :-
        createCardinality(N, 1, [0-2], List),
        assert(cardinalityList(List)),
        maplist(define_Cardinality, Rows),
        transpose(Rows, Columns),
        maplist(define_Cardinality, Columns).

define_Cardinality(X) :-
        cardinalityList(Lista),
        global_cardinality(X,Lista).
```

Figura 4 - predicado de restrição de cardinalidade

Na segunda restrição, que implica que entre duas células pretas seja respeitado o valor específico para essa linha/coluna é utilizado um autómato. Este autómato contém apenas 3 estados, em que o primeiro fica a aceitar todos os valores até encontrar um 0. Quando encontra esse valor, transita para o estado seguinte onde soma os valores das células até encontrar o 0 seguinte. Por fim, ao encontrar o último 0 transita para o estado de aceitação onde aceita os restantes valores da linha/coluna correspondente.

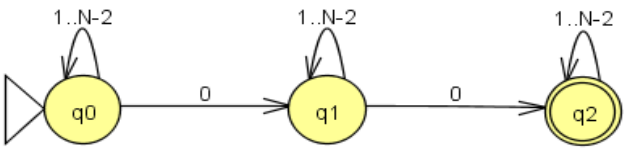


Figura 5 - autómato descrito

Esta abordagem permite que quando o autómato termina, a segunda restrição se encontre satisfeita. Assim, é aplicada uma abordagem através de restrições e não de geração e teste.

```
% Finished building automaton
sumArcs(0, Temp, Temp, _).

% Automaton state transitions needed to count the value of sums between black cells
sumArcs(N, Temp, List, Counter) :-
        N > 0,
        append(Temp, [arc(q0,N,q0), arc(q1, N, q1, [Counter + N]), arc(q2, N, q2)], NewList),
        NewN is N - 1,
        sumArcs(NewN, NewList, List, Counter).

% Restricts sum between black cells to a specific value
restrictSumInLine(Line, Value):-
        length(Line,N), NewN #= N-2,
        sumArcs(NewN, [arc(q0,0,q1), arc(q1,0,q2)], List, Counter),
        automaton(Line, _, Line, [source(q0), sink(q2)], List, [Counter], [0], [Value]).
```

Figura 6 - predicado de restrição por autómato

3.3 Estratégia de Pesquisa

A estratégia utilizada foi [bisect,down] que na maioria dos casos se mostrou mais eficiente que qualquer outra etiquetagem utilizada.

Esta abordagem implica que na escolha do valor das variáveis é feita uma escolha binária entre X # = <M e X #> M, onde M é o ponto médio do domínio de X. E este domínio é explorado em ordem decrescente. O que demonstra ser bastante mais eficiente devido ao facto de os valores no meio do domínio apresentarem maior probabilidade de apareceram do que os dos extremos.

4 Abordagem (Geração de problemas)

A abordagem na geração de problemas é praticamente idêntica à sua solução, porém, difere em alguns locais, os quais são salientados nas seguintes seções.

4.1 Restrições

A primeira restrição de cardinalidade mantém-se, sendo que, a segunda restrição é a que difere. O autómato utilizado garante que não existem células pretas consecutivas, não se preocupando com a soma entre elas pois não se trata da solução de problemas, mas sim a sua geração.

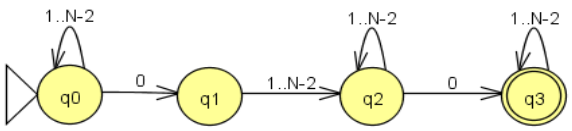


Figura 7 - autómato descrito

4.2 Função de avaliação

Embora na solução de problemas não tenha sido usada uma função de avaliação personalizada, para a sua geração foi. Esta função encontra todas as soluções possíveis dadas pelo labeling, na lista fdset_member, e realiza uma seleção aleatória das soluções.

```
% Select a random solution for generating a different board each attempt
myRandomSel(Var, _Rest, BB, BB1) :-
        fd_set(Var, Set),
        findall(Value, fdset_member(Value, Set), List),
        length(List, Len),
        random(0, Len, Number),
        nth0(Number, List, NewValue),
        (
           first_bound(BB, BB1), Var #= NewValue
        ;
           later_bound(BB, BB1), Var #\= NewValue
        ).
```

Figura 8 - predicado de seleção de solução aleatória

5 Utilização do programa

O programa pode ser começado através do predicado doppelBlock, o qual tem duas versões diferentes. A primeira opção, “doppelBlock.”, permite ao utilizador escolher tabuleiros fixos, que o programa contém, ou gerar um tabuleiro de tamanho 4 a 10. A segunda opção, “doppelBlock(+Doppel)”, permite utilizar um tabuleiro fornecido pelo utilizador, como por exemplo, “doppelBlock([[4,8,4,5,6,5],[9,7,2,10,3,1],_A]).”.

6 Visualização da Solução

O problema pode ser visualizado com o predicado “printDoppelPuzzle(+Doppel)”, e a sua solução pode ser visualizada com o predicado “printDoppelSolution(+Doppel, +Solution)”. Ambos os predicados mostram o problema como um tabuleiro quadrado em que existe um cabeçalho superior e lateral, para identificar as somas das linhas/colunas. As células pretas são representadas por pontos.

```
Doppelblock:
   | 4  8  4  5  6  5
--o------------------
9  |  .  .  .  .  .  .
7  |  .  .  .  .  .  .
2  |  .  .  .  .  .  .
10 |  .  .  .  .  .  .
3  |  .  .  .  .  .  .
1  |  .  .  .  .  .  .

Doppelblock solution:
   | 4  8  4  5  6  5
--o------------------
9  |  1  .  2  4  3  .
7  |  .  3  4  .  1  2
2  |  4  1  .  2  .  3
10 |  .  4  1  3  2  .
3  |  2  .  3  .  4  1
1  |  3  2  .  1  .  4
```

Figura 9 - exemplo de output para o tabuleiro de tamanho 6 incluído no programa

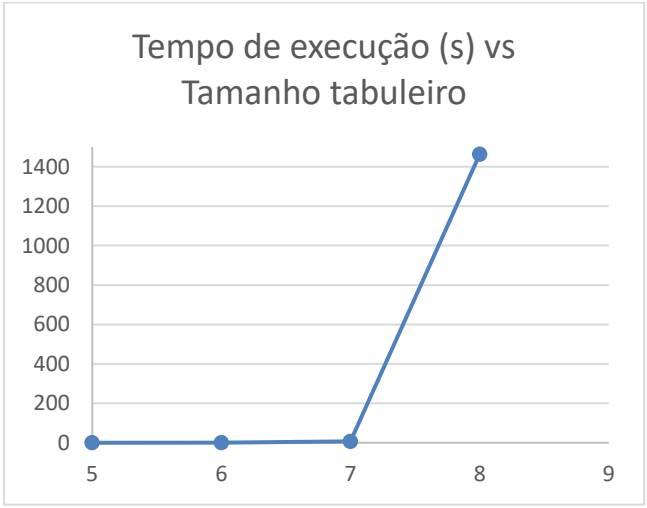
7 Resultados

Os testes realizados incidiram sobre o tempo de resolução de tabuleiros de tamanhos diferentes, bem como, resolução de tabuleiros fixos com variação das opções de labeling.

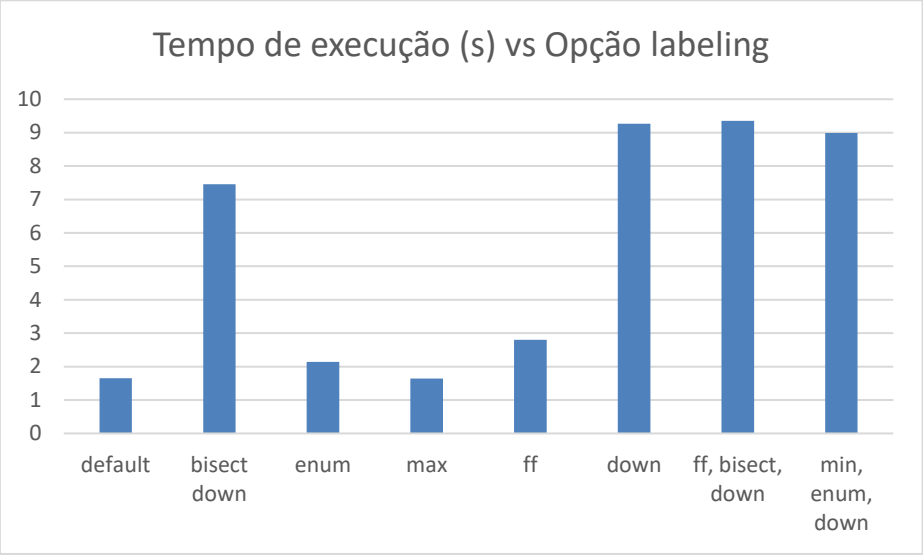
Num primeiro teste, verificou-se o tempo de execução para tabuleiros de tamanhos diferentes, a opção de labeling foi “bisect, down”.

```
boardS5([[1, 2, 3, 4, 4], [2, 3, 2, 5, 4], _]).
boardS6([[4, 8, 4, 5, 6, 5], [9, 7, 2, 10, 3, 1], _]).
boardS7([[4, 3, 4, 4, 5, 7, 15], [3, 5, 4, 5, 6, 3, 10], _]).
```

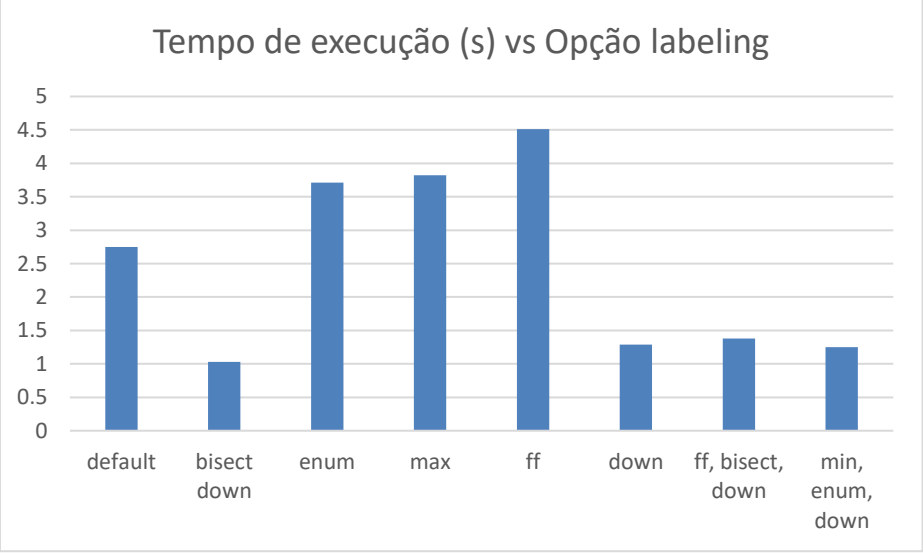
Figura 10 - tabuleiros testados para o primeiro teste



Num segundo teste, verificou-se o tempo de execução para diversas opções de labeling, o tabuleiro utilizado foi “doppelBlock([[4,3,4,4,5,7,15],[3,5,4,5,6,3,10],_]).”.



Num terceiro teste, repetiu-se o segundo teste mas para um tabuleiro 7x7 diferente, o qual foi “doppelBlock([[3,3,12,2,8,12,3],[2,8,5,3,3,7,3],_A]).”.



Os resultados obtidos permitem duas conclusões, primeiramente, o tamanho do tabuleiro escala o tempo de execução de forma exponencial. Por último, as opções de labeling não têm performance constante para o mesmo tamanho de tabuleiro, sendo que, a opção por defeito mostra a menor variação no tempo de execução.

8 Conclusões e Trabalho Futuro

A utilização de programação lógica por restrições permitiu concluir que é uma ferramente poderosa de resolução de problemas, que podem ser representados por domínios finitos, e por um conjunto de regras a cumprir.

Os resultados que se obtiveram permitem concluir que a solução desenvolvida tem custo exponencial à medida que o tamanho do tabuleiro aumenta, além disso, as opções de labeling apresentam uma performance variável para tabuleiros diferentes mas de igual tamanho, sendo que, a opção por defeito foi a que apresentou menos variação no tempo de execução.

A solução proposta resolve rapidamente tabuleiros de tamanho 4 a 7, porém tabuleiros de tamanho superior a 7 provam ser demasiado demorados a resolver. Um melhoramento da solução desenvolvida passaria, principalmente, por criar uma heurística de seleção personalizada, atendendo a alguns pontos, que são:

- Encontrar uma maneira de verificar se a opção de labeling por defeito seria mais eficaz que a escolhida, “bisect down”.
- Implementar uma função de seleção que experimentasse transpor o problema fornecido, visto que, em alguns casos houve diferença nos tempos de execução.
- Experimentar a utilização de timeouts e soluções não ideais.