Doppelblock

Relatório

Daniel Marques e João Carvalho

Faculdade de Engenharia da Universidade do Porto
Mestrado Integrado em Engenharia Informática e Computação
Programação em Lógica

Resumo O objetivo deste trabalho é implementar a resolução de um problema de decisão ou de otimização utilizando o paradigma de Programação em Lógica com Restrições. O desenvolvimento foi feito na linguagem Prolog com recurso ao sistema de desenvolvimento do SICStus Prolog.
O tema escolhido foi o Doppelblock, um puzzle que se enquadra num problema de decisão. O programa desenvolvido permite não só resolver este puzzle, como também gerá-lo. É possível indicar o tamanho, as somas e a matriz, sendo que todas estas possem ser variáveis não instanciadas, resultando em múltiplas soluções.

Keywords: doppelblock, sicstus, prolog, feup, mieic

1 Introdução

Este trabalho foi realizado no âmbito da Unidade Curricular de Programação em Lógica do Mestrado Integrado em Engenharia Informática e Computação da Faculdade de Engenharia da Universidade do Porto.

O objetivo do trabalho é implementar a resolução de um problema de decisão ou de otimização, tendo como inerente consequência proporcionar uma oportunidade para explorar paradigma de Programação em Lógica com Restrições consololidando importantes conceitos e técnicas específicas deste paradigma de programação.

O relatório surge em modo de documentação e análise do programa, seguindo a seguinte estrutura:

- **Descrição do Problema -** Descreve o problema em detalhe;
- **Abordagem -** Análise da modelação do problema;
  - **Variáveis de Decisão -** Descreve as variáveis de decisão e seus domínios;
  - **Restrições -** Descreve as restrições impostas às variáveis de decisão;
  - **Estratégia de Pesquisa -** Descreve a estratégia de etiquetagem.
- **Visualização da Solução -** Análise ilustrada dos predicados responsáveis pela visualização das soluções;
- **Resultados -** Demonstração de exemplos de solução de instâncias do problema, bem como outras capacidades do programa;
- **Conclusões -** Retrospetiva do trabalho realizado;
- **Bibliografia -** Recursos utilizados para desenvolver o programa;
- **Anexos -** Código fonte e modo de utilização.

2 Descrição do Problema

O puzzle Doppelblock é um problema de decisão que consiste numa matriz quadrada de tamanho n, com os seus elementos numerados de 1 a $n-2$ de forma a que em cada linha e coluna exatamente dois quadrados sejam pretos e cada número apareça uma única vez. Os números nas extremidades horizontais e verticais indicam a soma dos números entre os quadrados pretos nessa linha ou coluna, respetivamente.

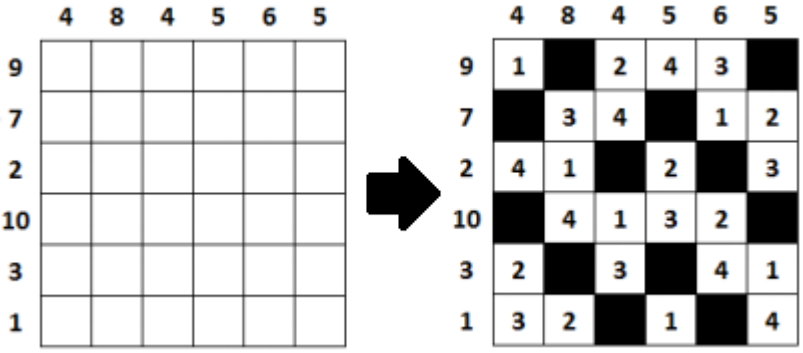


Figura 1. Exemplo de um puzzle de Doppelblock

3 Abordagem

Na implementação do puzzle em Prolog, o grupo representou a matriz como uma lista de listas, onde cada elemento é um número inteiro, sendo que os quadrados pretos são representados pelo valor zero. Em adição, as somas são representadas por duas listas de inteiros.

3.1 Variáveis de Decisão

A variável de decisão usada foi uma lista de listas de variáveis de domínio (inteiros) que nomeamos de *Board*. Esta variável representa a matriz de números e quadrados pretos. Esta variável é depois decomposta numa lista única de inteiros de modo a ser possível utilizar-se numa única chamada ao predicado *labeling/2*. Para cada lista do *Board* o domínio é definido e restrito para $[0, n-2]$, em que n é o tamanho da matriz.

3.2 Restrições

Quadrados pretos

Em cada linha e coluna da matriz têm de aparecer exatamente dois quadrados pretos que, em Prolog, atribuímos o valor *0* e que, na visualização em modo de texto, são representados pela letra 'X'. Esta restrição é assegurada quando o *Board* é inicializado no predicado *initBoardCycle/3*:

```
count(0, List, #=, 2)
```

Números distintos

Os números de uma linha e coluna são todos diferentes sendo que estes estarão no domínio $[1, n-2]$, dado que o valor *0* estará reservado para os quadrados pretos. Esta restrição é feita no predicado *setDistinctValues/3* que, para cada lista, define que um número apenas pode aparecer uma vez, utilizando, mais uma vez, o predicado *count/4*.

```
count(Counter, List, #=, 1)
```

Somas entre quadrados pretos

O somatório de todos os números comprimidos entre os dois quadrados pretos de uma linha ou coluna têm de ser iguais ao elemento correspondente na lista de somas lateral ou superior, respetivamente. Esta restrição é certificada pelo predicado *sumConstraint/2* que é invocado em cada linha e coluna da matriz. Este predicado determina a posição dos dois quadrados pretos e define que o somatório entre ambos é igual a soma correspondente.

```
sumConstraint(List, Sum) :-
  element(Black1, List, 0),
  element(Black2, List, 0),
  Black2 #> Black1,
  sumBetweenBlack(List, Sum, Black1, Black2).
```

O predicado *sumBetweenBlack/4* calcula o somatório entre ambos os quadrados pretos e define que este valor deve ser igual ao valor que recebe da lista de somas.

3.3 Estratégia de Pesquisa

A estratégia de etiquetagem utilizada foi a pré-definida do predicado *labeling/2* do SICStus Prolog. No entanto, existe um predicado principal que suporta a introdução de uma lista de opções que lhe é passada (*doppelblock/5*). Durante a implementação foi, de facto, experimentada a opção *ffc* que, em certas situações, torna a execução mais rápida. Contudo, era percetível que o desempenho era muito pior quando se requeria outra solução depois de uma já ter sido encontrada.

4 Visualização da Solução

Os predicados responsáveis pela visualização da solução estão definidos no ficheiro *display.pl*. Esta é feita em modo de texto, usando alguns caracteres que permitem a fácil e eficaz formatação de uma matriz. Primeiramente são desenhadas as somas superiores e depois a matriz, linha a linha, e, simultaneamente, as somas laterais. Em cada quadrado é chamado o predicado *drawCell/1* que o desenha dependendo de ser ou não um quadrado preto:

```
drawCell(0) :-
  write('X').
drawCell(Cell) :-
  write(Cell).
```

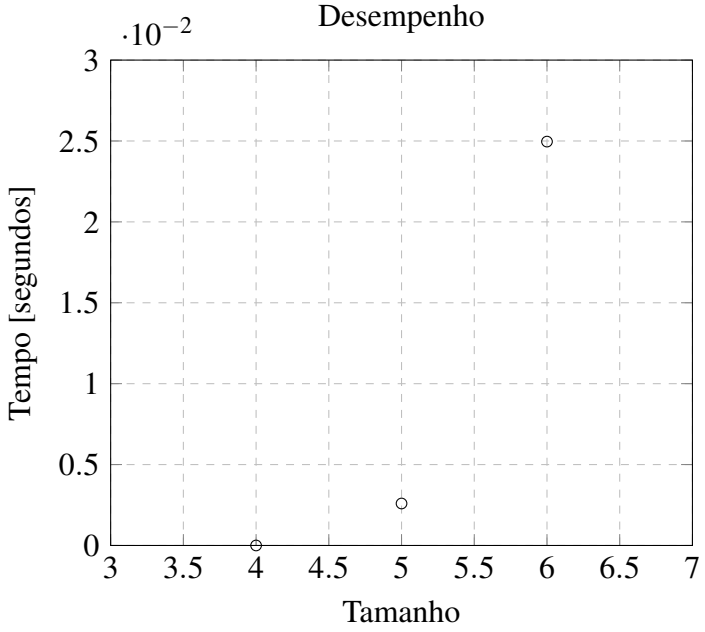
O resultado da instância do exemplo da *Figura 1* é o seguinte:

```
| ?- doppelblock(6,[4,8,4,5,6,5],[9,7,2,10,3,1],Board, ffc).
Solution found in: 0.735 seconds
     4     8     4     5     6     5
   _____ _____ _____ _____ _____ _____
  |     |     |     |     |     |     |
  |  1  |  X  |  2  |  4  |  3  |  X  | 9
  |_____|_____|_____|_____|_____|_____|
  |     |     |     |     |     |     |
  |  X  |  3  |  4  |  X  |  1  |  2  | 7
  |_____|_____|_____|_____|_____|_____|
  |     |     |     |     |     |     |
  |  4  |  1  |  X  |  2  |  X  |  3  | 2
  |_____|_____|_____|_____|_____|_____|
  |     |     |     |     |     |     |
  |  X  |  4  |  1  |  3  |  2  |  X  | 10
  |_____|_____|_____|_____|_____|_____|
  |     |     |     |     |     |     |
  |  2  |  X  |  3  |  X  |  4  |  1  | 3
  |_____|_____|_____|_____|_____|_____|
  |     |     |     |     |     |     |
  |  3  |  2  |  X  |  1  |  X  |  4  | 1
  |_____|_____|_____|_____|_____|_____|
Board = [[1,0,2,4,3,0],[0,3,4,0,1,2],[4,1,0,2,0,3],[0,4,1,3,2,0],[2,0,3,0,4,1],[3,2,0,1,0|...]] ? ;
no
```

Figura 2. Visualização em modo de texto.

5 Resultados

A abordagem ao problema foi desenvolvida a pensar no melhor desempenho do programa, no entanto a complexidade do problema sobrepõe-se e, consequentemente, o tempo de execução aumenta drasticamente com o tamanho da matriz. A conclusão a que chegamos foi que o problema é NP-Completo, bem como muitos outros puzzles que constituem problemas de decisão, tal como o Sudoku. No gráfico seguinte estão representados vários exemplos e o seu tamanho e tempo de execução correspondentes.



Analisando o gráfico podemos concluir que, efetivamente, o tempo de execução aumenta exponencialmente com o tamanho da matriz.

6 Conclusões

Com o desenvolvimento deste projeto, concluiu-se, mais uma vez, que o Prolog é uma linguagem muito poderosa e eficiente para a resolução de problemas lógicos. O desenvolvimento de programas complexos é simplificado com o uso da biblioteca *clpfd* e a compreensão dos predicados lá definidos, assim como a correta abordagem ao problema, definindo variavéis de decisão, seus domínios e restrições. Concluímos que esta biblioteca e o seu paradigma de Programação em Lógica com Restrições tem um enorme potêncial na resolução de problemas de decisão e otimização. Na nossa implementação, tentamos melhorar ao máximo o desempenho do programa, no entanto haverá certamente aspetos a melhorar.

Referências

1. Regras, http://logicmastersindia.com/lmitests/dl.asp?attachmentid=659&view=1
2. Exemplos, https://www.janko.at/Raetsel/Doppelblock/index.html
3. Documentação, https://sicstus.sics.se/

Anexos

Modo de utilização

Os predicados de entrada no programa são *doppelblock/4* e *doppelblock/5*. Para mais informações execute o predicado *printUsage/0*.

Código fonte

O código fonte está listado abaixo e também se encontra disponível comprimido no seguinte url: https://www.dropbox.com/s/mw31oor40jjevtj/plog_src.zip?dl=0.

```
/*
  doppelblock.pl

  This file holds the predicates that are entry points to the solver.
*/

:- use_module(library(timeout)).
:- include('board.pl').
:- include('display.pl').


/*
  doppelblock(?Size, ?TopSums, ?RightSums, ?Board)
    @brief Solves an instance of the puzzle

    @param Size - The size of the matrix
  @param TopSums - The sums on top of the board
  @param RightSums - The sums on the right of the board
  @param Board - The matrix (inner board)

    Example: doppelblock(6, [4,8,4,5,6,5], [9,7,2,10,3,1], Board).
*/
doppelblock(Size, TopSums, RightSums, Board) :- !,
  doppelblock(Size, TopSums, RightSums, Board, []).
doppelblock(_, _, _, _) :-
  printUsage, !.


/*
  doppelblock(?Size, ?TopSums, ?RightSums, ?Board, +LabelingOptions)
    @brief Solves an instance of the puzzle
```

```
    @param Size - The size of the matrix
  @param TopSums - The sums on top of the board
  @param RightSums - The sums on the right of the board
  @param Board - The matrix (inner board)
  @param LabelingOptions - Option list for the labeling/2 predicate

    Example: doppelblock(6, [4,8,4,5,6,5], [9,7,2,10,3,1], Board, [ffc]).
*/
doppelblock(Size, TopSums, RightSums, Board, LabelingOptions) :- !,
  statistics(walltime, [T0|_]),
  checkArguments(Size, TopSums, RightSums),
  initBoard(Board, Size),
  transpose(Board, TransposeBoard),
  initBoard(TransposeBoard, Size),
  applySumsConstrains(Board, RightSums),
  applySumsConstrains(TransposeBoard, TopSums),
  label(Board, LabelingOptions),
  statistics(walltime, [T1|_]),
  ElapsedTime is T1 - T0,
  format('Solution found in: ~6d seconds', [ElapsedTime]), nl,
  fd_statistics,
  drawBoard(Board, TopSums, RightSums, Size).
doppelblock(_, _, _, _, _) :-
  printUsage, !.


/*
  printAllSolutions(+OutputFile, +Time, -Result, ?Game)
    @brief Writes all the solutions generated during a timeout

    @param OutputFile - The file to write the results to
  @param Time - Timeout in milliseconds (the predicates end once it is
      reached)
  @param Result - The timout result
  @param Game - A list representing an instance of the game

    Example: printAllSolutions('solutions.txt', 5000, Result, [6, _, _, _]).
*/
printAllSolutions(OutputFile, Time, Result, [Size, TopSums, RightSums, Board
    ]) :-
  open(OutputFile, write, X),
  current_output(CO),
  set_output(X),
  printAllSolutions(Time, Result, [Size, TopSums, RightSums, Board]),
  close(X),
  set_output(CO).
printAllSolutions(Time, Result, [Size, TopSums, RightSums, Board]) :-
  time_out((doppelblock(Size, TopSums, RightSums, Board), fail), Time, Result
      ).
printAllSolutions(_, _, _).
```

```
/*
  board.pl

  This file is responsible for applying the domain of each variable and its
      restrictions.
*/

:- use_module(library(clpfd)).
:- use_module(library(lists)).
:- include('utilities.pl').


/*
  initBoard(+Board, +Size, +Counter)
    @brief Initializes the Board, its domain and restrictions

  @param Board - The matrix (inner board)
  @param Size - The size of the matrix
*/
initBoard(Board, Size) :-
  length(Board, Size),
  initBoardCycle(Board, Size, 0).


/*
  initBoardCycle(+Board, +Size, +Counter)
    @brief Initializes the Board, its domain and restrictions

  @param Board - The matrix (inner board)
  @param Size - The size of the matrix
  @param Counter - A counter to initialize all rows
*/
initBoardCycle(Board, Size, Counter) :-
  Counter < Size,
  length(List, Size),
  Domain is Size - 2,
  domain(List, 0, Domain),
  count(0, List, #=, 2),
  setDistinctValues(List, Domain, 1),
  nth0(Counter, Board, List),
  Ncounter is Counter + 1, !,
  initBoardCycle(Board, Size, Ncounter).
initBoardCycle(_,_,_).


/*
  setDistinctValues(+List, +Domain, +Counter)
    @brief Applies the constraint that defines that all numbers in a row/
        column must be different

  @param List - A row or column to apply the constraint to
  @param Domain - The maximum number of each list (Size - 2)
  @param Counter - A counter that starts at 1 and ends at Domain
```

```
*/
setDistinctValues(List, Domain, Counter) :-
  Counter =< Domain,
  count(Counter, List, #=, 1),
  NewCounter is Counter + 1,
  setDistinctValues(List, Domain, NewCounter).
setDistinctValues(_, _, _).


/*
  applySumsConstrains(+Board, +SumList)
    @brief Applies the sum constraint to the board in the horizontal
        direction

  @param Board - The matrix (inner board)
  @param SumList - A list of sums
*/
applySumsConstrains([],[]):- !.
applySumsConstrains([BoardH|BoardT], [SumH|SumT]) :-
  sumConstraint(BoardH, SumH),
  applySumsConstrains(BoardT, SumT).


/*
  sumBetweenBlack(+List, +Sum, +Black1, +Black2)
    @brief Applies the sum constraint to a List (the sum of all elements
        between Black1 and Black2 must be Sum)

  @param List - The list to apply the constraint
  @param Black1 - The position of the first black cell in the List
  @param Black2 - The position of the second black cell in the List
  @param Sum - The expected sum
*/
sumBetweenBlack(List, Sum, Black1, Black2) :-
  Counter #= Black1 + 1,!,
  sumBetweenBlack(List, Sum, 0, Black1, Black2, Counter).
sumBetweenBlack(_, Sum, TmpSum, _, Black2, Counter) :-
  Sum #= TmpSum,
  Black2 #= Counter.
sumBetweenBlack(List, Sum, TmpSum, Black1, Black2, Counter) :-
  element(Counter, List, Element),
  NewTmpSum #= TmpSum + Element,
  NewTmpSum #=< Sum,
  NewCounter #= Counter + 1, !,
  sumBetweenBlack(List, Sum, NewTmpSum, Black1, Black2, NewCounter).


/*
  sumConstraint(+List, +Sum)
    @brief Applies the sum constraint to a List (the sum of all elements
        between black cells must be Sum)

  @param List - A row or column to apply the constraint to
```

```
  @param Sum - The expected sum
*/
sumConstraint(List, Sum) :-
  element(Black1, List, 0),
  element(Black2, List, 0),
  Black2 #> Black1, !,
  sumBetweenBlack(List, Sum, Black1, Black2).
```

board.pl

```
/*
  display.pl

  This file is responsible for the console interface of the program.
*/


/*
  drawBoard(+Board, +TopSums, +RightSums, +Size)
    @brief Prints the board to the screen

  @param Board - The matrix (inner board)
  @param TopSums - The sums on top of the board
  @param RightSums - The sums on the right of the board
  @param Size - The size of the matrix
*/
drawBoard(Board, TopSums, RightSums, Size) :-
  write('   '), drawTopSums(TopSums), nl,
  write('   '), drawHeader(Size), nl,
  drawBoard(Board, RightSums, Size), !.
drawBoard([], _, _).
drawBoard([Row | Rest], [Sum | OtherSums], Size) :-
  drawUpSeparatorLine(Size),
  write('  |'),
  drawRow(Row),
  write('  '),
  write(Sum), nl,
  drawDownSeparatorLine(Size),
  drawBoard(Rest, OtherSums, Size).


/*
  drawRow(+Row)
    @brief Prints a row of the board

  @param Row - A row of the board
*/
drawRow([]).
drawRow([Cell | Remainder]) :-
  write('  '),
  drawCell(Cell),
  write('  |'),
  drawRow(Remainder).
```

```
/*
  drawUpSeparatorLine(+Size)
    @brief Prints the upper separator between rows

  @param Size - The size of the matrix
*/
drawUpSeparatorLine(Size) :-
  write('   |'),
  drawUpSeparatorLine(Size, Size), nl.
drawUpSeparatorLine(_, 0).
drawUpSeparatorLine(Size, Counter) :-
  write('      |'),
  NextCounter is Counter - 1,
  drawUpSeparatorLine(Size, NextCounter).


/*
  drawDownSeparatorLine(+Size)
    @brief Prints the bottom separator between rows

  @param Size - The size of the matrix
*/
drawDownSeparatorLine(Size) :-
  write('   |'),
  drawDownSeparatorLine(Size, Size), nl.
drawDownSeparatorLine(_, 0).
drawDownSeparatorLine(Size, Counter) :-
  write('-----|'),
  NextCounter is Counter - 1,
  drawDownSeparatorLine(Size, NextCounter).


/*
  drawHeader(+Size)
    @brief Prints the separator between the TopSums and the matrix

  @param Size - The size of the matrix
*/
drawHeader(Size) :-
  drawHeader(Size, Size).
drawHeader(_, 0).
drawHeader(Size, Counter) :-
  write('----- '),
  NextCounter is Counter - 1,
  drawHeader(Size, NextCounter).


/*
  drawTopSums(+TopSums)
    @brief Prints the top sums

  @param TopSums - The sums on top of the board
```

```
*/
drawTopSums([]).
drawTopSums([Sum | Rest]) :-
  write('  '), write(Sum), write('   '),
  drawTopSums(Rest).


/*
  drawCell(+Cell)
    @brief Prints a cell. If the cell is zero, it is considered a black cell.

  @param Cell - A matrix element
*/
drawCell(0) :-
  write('X').
drawCell(Cell) :-
  write(Cell).

/*
  printUsage
    @brief Prints the usage in case the arguments are wrong.
*/
printUsage :-
  write('Usage: doppelblock(?Size, ?TopSums, ?RightSums, ?Board, [+
      LabelingOptions]).'), nl,
  write('  Size            - Integer      - The Size of the Board'), nl,
  write('  TopSums         - Integer List - The sums on top of the board'),
      nl,
  write('  RightSums       - Integer List - The sums on the left of the board
      '), nl,
  write('  Board           - Integer List - The Board correspondent to the
      sums'), nl,
  write('  LabelingOptions - Atom List    - Options to labeling the variables
      '), nl.
```

display.pl

```
/*
  utilities.pl

  This file is responsible for supplying utility predicates.
*/


/*
  label(+Board, +LabelingOptions)
    @brief Labels the board variables

  @param Board - The matrix (inner board)
  @param LabelingOptions - Option list for the labeling/2 predicate
*/
label(Board, LabelingOptions) :-
  flattenBoard(Board, Label),!,
  labeling(LabelingOptions, Label).
```

```
/*
  flattenBoard(+Board, -Flat)
    @brief Takes a board (list of lists) and returns a single list containing
         all variables

  @param Board - The matrix (inner board)
  @param Flat - A list containing all variables from Board
*/
flattenBoard(Board, Flat) :-
  flattenBoard(Board, Flat, []).
flattenBoard([], Flat, Flat).
flattenBoard([Row | Rest], Flat, TmpFlat) :-
  append(TmpFlat, Row, NewTmpFlat),
  flattenBoard(Rest, Flat, NewTmpFlat).


/*
  checkArguments(?Size, ?TopSums, ?RightSums)
    @brief Checks if the arguments provided are valid

    @param Size - The size of the matrix
  @param TopSums - The sums on top of the board
  @param RightSums - The sums on the right of the board
*/
checkArguments(Size, TopSums, LeftSums) :-
  length(TopSums, Size),
  length(LeftSums, Size),
  Size > 1.
```

utilities.pl