Resolução de Problema de Decisão usando Programação em Lógica com Restrições: MNO puzzle

José António Fonseca Guerra e Martim Pinto da Silva
FEUP-PLOG, Turma 3MIEIC05, Grupo MNO Puzzle_3

Faculdade de Engenharia da Universidade do Porto
Rua Dr. Roberto Frias, 4200-465, Porto, Portugal

Resumo. O projeto foi desenvolvido no Sistema de Desenvolvimento SICStus Prolog, no âmbito da unidade curricular de Programação em Lógica, cujo objetivo é resolver um problema de decisão implementando restrições. O problema de decisão escolhido foi o MNO puzzle , este, tem como finalidade obter um tabuleiro seguindo um conjunto de regras descrito neste artigo. Assim, através da linguagem de Prolog, foi possível resolver este problema de forma eficiente e efetiva.
Keywords: MNO Puzzle, Programação de Restrições Lógicas, Sicstus, Prolog.

1. Introdução

O projeto foi desenvolvido no âmbito da unidade curricular de Programação em Lógica de 3º ano do curso Mestrado Integrado em Engenharia Informática e de Computação. Para tal, foi necessário implementar uma possível resolução para um problema de um puzzle, com restrições sendo que o nosso grupo optou pelo jogo chamado MNO puzzle.

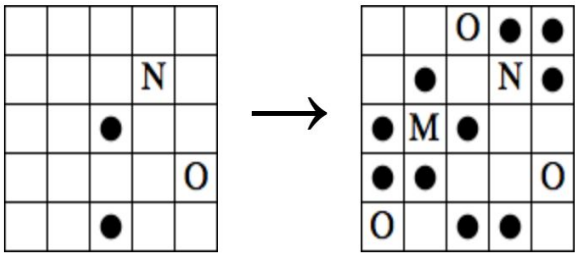
Este artigo segue a seguinte estrutura:

- **Descrição do Problema**: descrição com detalhe do problema de otimização ou decisão em análise.
- **Abordagem**: descrição da modelação do problema como um PSR / POR, de acordo com as subsecções seguintes:
    - **Variáveis de Decisão**: descrição das variáveis de decisão e dos seus domínios, e do seu significado no contexto do problema em análise.
    - **Restrições**: descrição das restrições rígidas e flexíveis do problema e a sua implementação utilizando o SICStus Prolog.
    - **Função de Avaliação**: descrição da forma de avaliar a solução obtida e a sua implementação utilizando o SICStus Prolog.
    - **Estratégia de Pesquisa**: descrição da estratégia de etiquetagem (labeling) utilizada ou implementada, nomeadamente heurísticas de ordenação de variáveis e valores.
- **Visualização da Solução**: explicação dos predicados que permitem visualizar a solução em modo de texto.
- **Resultados**: exemplos de aplicação em instâncias do problema com diferentes dimensões e análise dos resultados obtidos.
- **Conclusões e Trabalho Futuro**: Que conclusões retira deste projeto? O que mostram os resultados obtidos? Quais as vantagens e limitações da solução proposta? Como poderia melhorar o trabalho desenvolvido?
- **Bibliografia**: Livros, artigos, páginas Web, usados para desenvolver o trabalho.
- **Anexos**: Código fonte, ficheiros de dados e resultados.

2. Descrição do problema

O problema escolhido consiste em resolver um tabuleiro N*N, já parcialmente preenchido, com um conjunto de regras abaixo especificadas. A dificuldade do puzzle pode variar conforme o tamanho do puzzle e o número de elementos inicialmente gerados.

1. Cada linha e cada coluna tem de conter dois pontos e uma letra, havendo só 3 tipos de letras.
2. Para ser letra M (midpoint), a letra tem de estar entre 2 pontos e ambos têm de estar à mesma distância de M, para a coluna e linha em que se insere.
3. Para ser letra N, a letra tem de estar entre 2 pontos e ambos têm de estar a uma distância diferente de N, para a coluna e linha em que se insere.
4. Para ser letra O, a letra não pode estar entre dois pontos, para a coluna e linha em que se insere.



3. Abordagem

Na resolução deste problema, na linguagem *Prolog,* foi utilizada uma lista de listas para representar o tabuleiro do jogo, sendo que cada elemento das listas é um número inteiro que varia entre 0 e 4, ou '_' (caso o elemento seja desconhecido) e representa o valor do elemento (ponto, espaço vazio, letras M,N e O) e a biblioteca do sicstus de clpfd (programação de restrições lógicas sobre domínios finitos).

Relativamente à implementação das restrições optámos por dividir o problema em dois. Em primeiro lugar, recebendo o tabuleiro como uma lista de listas fazemos as restrições para cada linha do puzzle. Em seguida, com auxílio ao predicado transpose, é calculada a transposta da matriz recebida e voltamos a chamar o predicado que restringe as linhas, desta vez restringindo as colunas.

```
% Restrictions for lines
solveMatrix(Matrix),
% Restrictions for columns
transpose(Matrix, TMatrix),
solveMatrix(TMatrix),
```

Para isso, elaboramos um autômato que restringe não só o número de pontos e letras para cada linha, como também o tipo de letra para qualquer posição(ver abaixo).

3.1 Variáveis de decisão

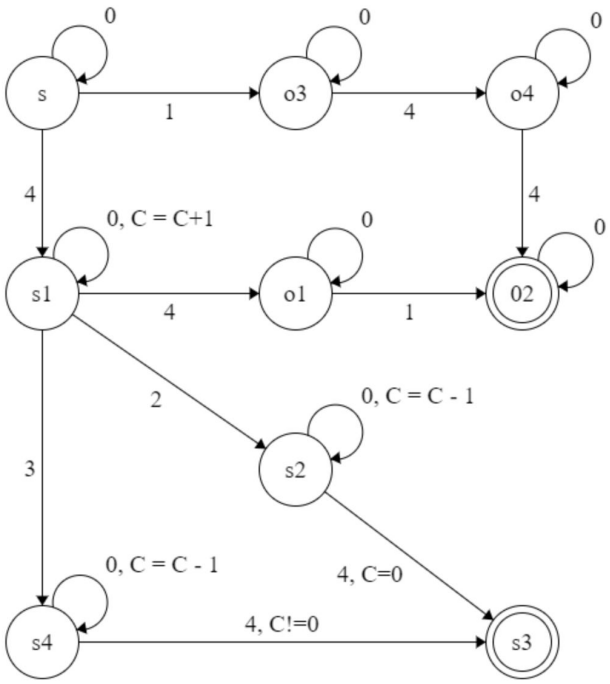
As variáveis de decisão associadas à resolução deste problema são: o número de pontos por linha, o número de pontos por coluna, as restrições apliacaa???

O domínio dos elementos do puzzle variam entre 0 e 4, representando, respectivamente, um espaço vazio, a letra O, a letra M, a letra N e um ponto.

```
% Specifies the domain of the matrix
domain(OneListMatrix, 0, 4),
```

3.2 Restrições

Todas as restrições impostas são implementadas com o uso de um automata portanto é necessário proceder com a explicação detalhada de como este automata, representado na figura em baixo, é capaz de validar às linhas e às colunas do nosso puzzle.



O estado s é estado inicial e os estados o2 e s3 são os estados finais. Os estados S1, S2, S3 e S4 servem para validar linhas ou colunas em que existam a letra M ou N, enquanto que os estados o1, o2, o3, o4 servem para validar linhas ou colunas em que a letra O exista.

O automata começa por no

7 Anexos

7.1 Tabelas e Gráficos

7.2 Código fonte

lib.pl

```
:-use_module(library(clpfd)).
:-use_module(library(random)).

reset_timer :-
    statistics(walltime, _).

print_time :-
    statistics(walltime,[_,T]),
    TS is ((T // 10) * 10) / 1000,
    nl, write('Solution Time: '), write(TS), write('s'), nl, nl.

clear :-
    clear_console(100), !.

clear_console(0).
clear_console(N) :-
    nl,
    N1 is N-1,
    clear_console(N1).
```

board.pl

```
%   Conversion between what is stored and displayed
piece(0, ' ').
piece(1, 'O').
piece(2, 'M').
piece(3, 'N').
piece(4, '*').
piece(_, ' ').

%   Prints the top part of the board
printBoardTop(0):-
    write('\n').
printBoardTop(Counter) :-
    write(' _ _ _'),
    CounterAux is Counter - 1,
    printBoardTop(CounterAux).

%   Prints the down part of the board
printBoardDown(0):-
    write('|\n').
printBoardDown(Counter) :-
    write('|_ _ _'),
    CounterAux is Counter - 1,
    printBoardDown(CounterAux).
```

```
%   Prints the contents of a line
printBoardBody([], N, N).
printBoardBody([H|T], N, Line) :-
    %   Iterates through the rows of the board
    printBarsRows(N),
    write('|  '),
    printBoardLine(H, Line),
    printBoardDown(N),

    LineI is Line+1,
    printBoardBody(T, N, LineI).

%   Prints the board sent in variable X
printBoard(X, N):-
    printBoardTop(N),
    printBoardBody(X, N, 0).
```

```
%  Prints the rows of the board
printBarsRows(0):-
    write('|\n').
printBarsRows(Counter):-
    write('|     '),
    CounterAux is Counter - 1,
    printBarsRows(CounterAux).

%  Prints each line of the board line content
printBoardLine([], _):-
    write('\n').
printBoardLine([H|T], Line) :-
    piece(H, S),
    write(S),
    write('  |  '),
    printBoardLine(T, Line).
```

Solver.pl

```
:-use_module(library(lists)).
:-use_module(library(between)).
:-use_module(library(system)).
:-[board].
:-[lib].
:-[puzzles].

solver(Matrix, N) :-
    % Resets time for statistics
    reset_timer,
    % Generates a board bazed on predefined boards
    generate_board(N, Selected, Matrix),
    once(printBoard(Selected, N)),
    sleep(2),
    % Stores the matrix as 1 dimension
    append(Matrix, OneListMatrix),
    % Specifies the domain of the matrix
    domain(OneListMatrix, 0, 4),
    % Restrictions for lines
    solveMatrix(Matrix),
    % Restrictions for columns
    transpose(Matrix, TMatrix),
    solveMatrix(TMatrix),
    % labeling of the one list matrix
    !, labeling([], OneListMatrix),
    % prints the game Board
    once(printBoard(Matrix, N)),
    % prints elapsed time
    print_time.
```

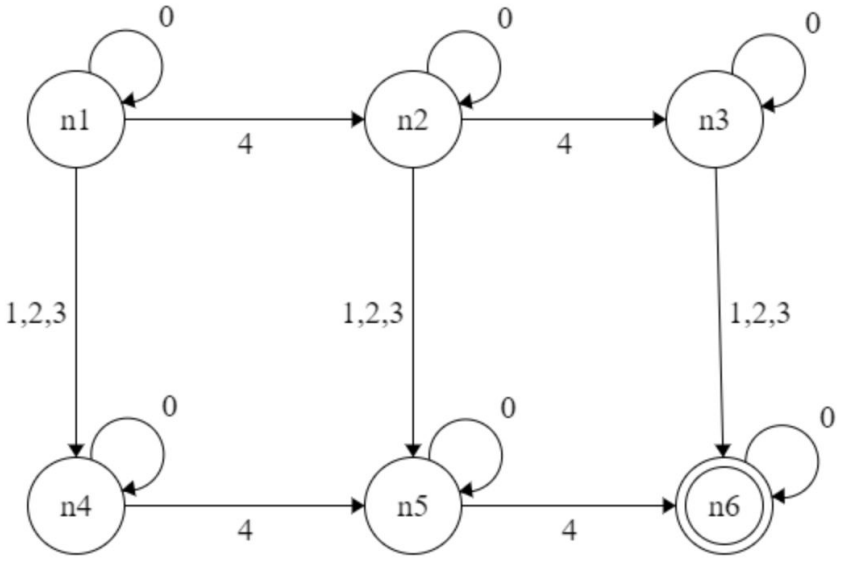
```
% Solves received matrix (solves only the lines)
solveMatrix([]).
solveMatrix([H|T]) :-
    length(H, LineLength),
    solveLine(H, 0, LineLength),
    solveMatrix(T).

% Restricts each line (2 dots and 1 letter per line + restricts line)
solveLine(_, LineLength, LineLength).
solveLine(Line, Index, LineLength) :-
    nth0(Index, Line, IndexElem),
    piece(M, 'M'),
    piece(N, 'N'),
    piece(O, 'O'),
    piece(Dot, '*'),
    piece(Null, ' '),

    allPointsAutomata(Line),

    NextIndex is Index + 1,
    solveLine(Line, NextIndex, LineLength).
```

```
% Automaton for M,N and O points
allPointsAutomata(Line) :-
    automaton(Line, _, Line, [source(s), sink(s3), sink(o
        [
            arc(s, 0, s),
```


0
0
0
n1
4
n2
4
n3
1,2,3
1,2,3
1,2,3
0
0
0
n4
4
n5
4
n6