

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

José António 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 optou-se 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 $N \times N$, 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 M, N e O.
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.

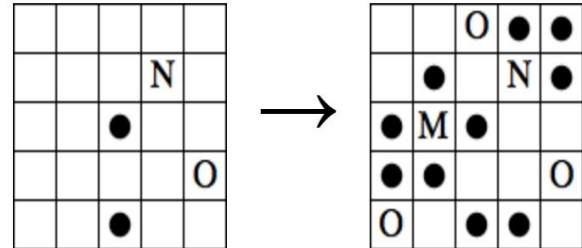


Figura 1 – Solução de um puzzle MNO

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/2**, é calculada a transposta da matriz recebida e voltamos a chamar o predicado que restringe as linhas, desta vez restringindo as colunas.

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).

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

Figura 2 – Predicado solveMatrix

3.1 Variáveis de decisão

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

Figura 3 – Aplicação do domínio

As variáveis de decisão associadas à resolução deste problema são: o número de pontos por linha e coluna, o número de letras por linha e coluna e as distâncias dos pontos às letras M e N. Muitas das variáveis de decisão são automatizadas pelo autômato que foi criado para a solução do problema. 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.

3.2 Restrições

```

solveMatrix([], _).
solveMatrix([H|T], N) :-
    automaton(H, _, H, [source(s), sink(s3), sink(o2)],
    [
        arc(s, 0, s),
        arc(s, 1, o3),
        arc(s, 4, s1),

        arc(s1, 0, s1, [C+1]),
        arc(s1, 2, s2),
        arc(s1, 3, s4),
        arc(s1, 4, o1),

        arc(s2, 0, s2, [C-1]),
        arc(s2, 0, s5),
        arc(s2, 4, s3),

        arc(o1, 0, o1),
        arc(o1, 1, o2),

        arc(o2, 0, o2),

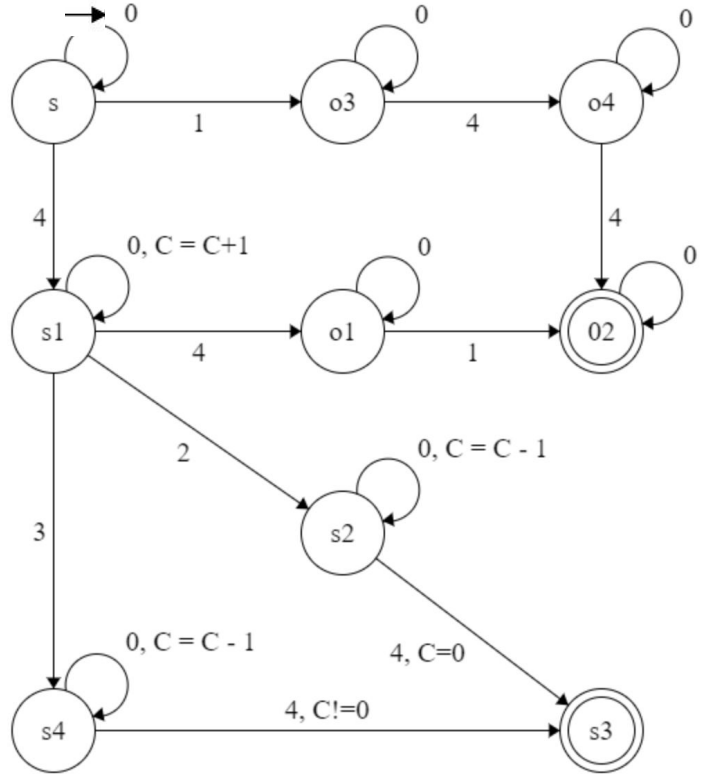
        arc(o3, 0, o3),
        arc(o3, 4, o4),

        arc(o4, 0, o4),
        arc(o4, 4, o2),

        arc(s3, 0, s3),

        arc(s4, 0, s4, [C-1]),
        arc(s4, 4, s3)
    ],
    [C], [0], [DeltaDist]
),
    Naux is N - 1,
    solveLine(H, DeltaDist, 0, Naux),
    solveMatrix(T, N).

```



Todas as restrições impostas são implementadas com o uso de um automata (NFA) representado na figura acima. É responsável por garantir que por cada linha e coluna hajam 2 pontos e 1 letra. Além disso, garante que as letras M e N estão entre 2 pontos e a letra O está fora dos pontos (i.e não está entre 2 pontos).

O estado S é o 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 tenham 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.

Para implementar a restrição $C \neq 0$ quando se transita do estado s4 para s3 e $C = 0$ quando se transita de s2 para s3 foi necessário desenvolver o predicado **solveLine/4** quer percorre uma linha e que impõe restrições para o valor final da variável C que é o único contador do automata. Este contador é a variação da distância entre os 2 pontos e a letra, ou seja, se $C=0$ trata-se da letra M e se $C \neq 0$ trata-se de uma letra N.

Caso o elemento esteja num dos extremos de uma linha ou coluna então não é necessário impor nenhuma restrição já que o autômato já faz isso, por essa razão faz-se essa verificação com um condicional. Caso o elemento não esteja nos extremos então caso seja uma letra M então quer dizer que o contador do autômato tem quer igual a

zero, isto é, a distância dos pontos a letra M tem que ser a mesma. Caso seja uma letra N então quer dizer que o contador do autômato tem que ser diferente de zero, isto é a distância dos pontos a letra N é diferente.

```
% Restricts each line (2 dots and 1 letter per line + restricts line)
solveLine([], _, _, _).
solveLine([H | T], DeltaDist, Counter, Length):-
(
    (Counter \=0, Counter \= Length) ->
    fd_batch([DeltaDist #= 0 #<= H #= 2,DeltaDist #\=0 #<= H #= 3]); true
),
    CounterAux is Counter + 1,
    solveLine(T, DeltaDist, CounterAux, Length).
```

3.3 Função de Avaliação

Este problema não é propriamente um problema de otimização em que tem-se de encontrar a melhor solução, para cada tabuleiro apenas existe uma solução logo as únicas formas de avaliação que se tem do desempenho do código é através do tempo que o programa demora a colocar todas as restrições e a fazer labelling e das Estatísticas de execução específicas do solver clp(fd) (resumptions, entailments, prunings, backtracks, constraints). **Resumptions** são o número de vezes que uma restrição foi reatada, os **entailments** são o número de vezes que um ('dis')entailment foi detectado, os **prunings** são o número de vezes que um domínio foi reduzido, os **backtracks** são o número de vezes que foi encontrada uma contradição por um domínio ter ficado vazio ou uma restrição global ter falhado e finalmente os **constraints** são o número de restrições criadas.

Para contar o tempo utiliza-se os predicados *reset_timer/0* e *print_time/0*. O *reset_timer/0*, tal como o nome indica, redefine o contador do tempo de execução para 0 com a ajuda do predicado **statistics/2**. O predicado *print_time/0* escreve para a consola o tempo em segundos com duas casas decimais. Quando se quer medir o tempo de execução é apenas necessário antes do bloco de código executar o predicado **reset_timer/0** e depois quando o bocado de código que queremos contar o tempo acabar, chamar o predicado **print_time/0** para imprimir o tempo decorrido.

Para imprimir as estatísticas de execução específicas do solver clp(fd) é apenas preciso fazer a chamada ao predicado *fd_statistics/0* que é chamado dentro e no final do predicado *print_time/0* já que sempre que se quer imprimir o tempo também se quer ver as estatísticas até aquele momento.

```
reset_timer :-
    statistics(walltime, _).

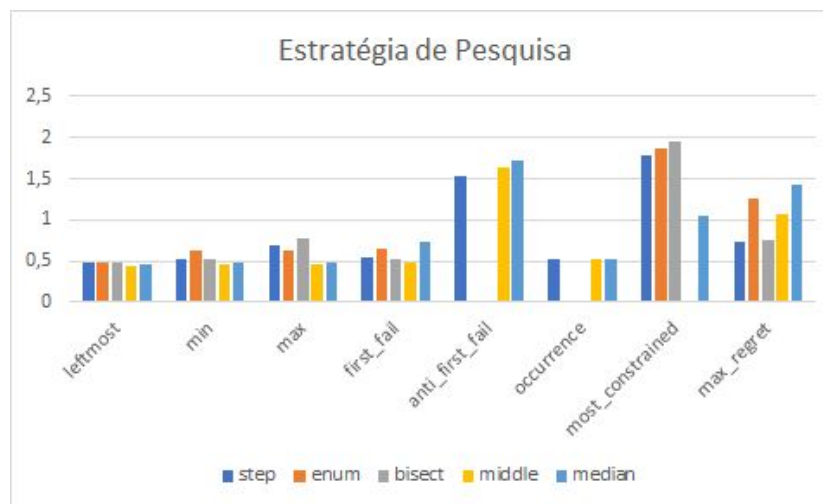
print_time :-
    statistics(walltime, [_, T]),
    TS is ((T // 10) * 10) / 1000,
    nl, write('Solution Time: '), write(TS), write('s'), nl
    fd_statistics.
```

```
Solution Time: 0.01s

Resumptions: 3858
Entailments: 746
Prunings: 2839
Backtracks: 1
Constraints created: 622
```

3.4 Estratégia de Pesquisa

Foram testadas várias opções de pesquisa através dos valores que se foram passando no primeiro parâmetro do predicado **labelling/2**. Ao nível da ordenação das variáveis, no labelling, para se saber como seleccionar a próxima variável passou-se no primeiro argumento um dos seguintes valores: **leftmost**, **min**, **max**, **first_fail**, **anti_first_fail**, **occurrence**, **most_constrained**, **max_regret**. Ao nível da selecção de valores, no labelling, para se saber que valores seleccionar os valores para uma variável passou-se no primeiro argumento um dos seguintes valores: **step**, **enum**, **bisect**, **median/middle**. Em suma, no labelling foi passado no primeiro argumento sempre um par de valores resultantes da combinação de um valor da parte da ordenação de variáveis e um valor da parte da selecção de valores. Todas as combinações destes valores bem como o tempo resultante da execução de cada labelling com estes valores pode ser encontrado na tabela seguinte:.

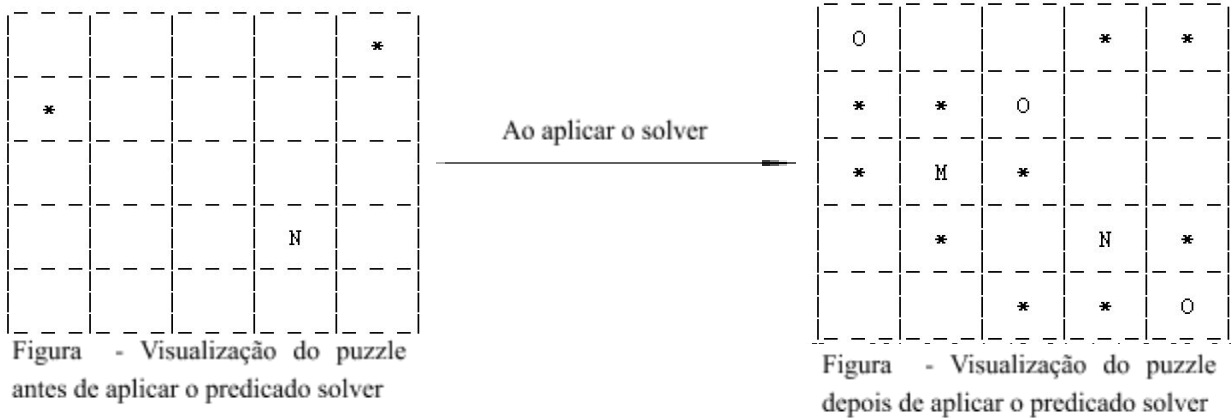


Os tempos obtidos nesta tabela são para a resolução de um tabuleiro 40x40 que se encontra totalmente vazio. Com esta tabela podemos concluir que qualquer combinação de valores de selecção de valores com os valores de ordenação de variáveis **max**, **min**, **left_most** e **first_fail** providenciam resultados muito parecidos e razoavelmente bons em termos de tempos. Por outro lado qualquer combinação de valores de selecção de valores com os valores de ordenação de variáveis **anti_first_fail**, **occurrence**, **most_constrained**, **max_regret** conduziam a um resultado pior em termos de tempos.

Nota: Mais informação sobre este tema pode ser encontrada na secção de Resultados

4 Visualização da solução

A visualização da solução é feita com recurso a predicados do ficheiro **board.pl** através dos quais imprime-se o tabuleiro de jogo inicialmente gerado e o puzzle resolvido. O predicado responsável pela apresentação do puzzle é o *printBoard/2* que recebe uma lista de listas e o tamanho de cada lista. Esse predicado também usa outros predicados de ajuda como o *printBoardTop/1*, *printBoadDown/1*, *printBarsRows/1*, *printBoardLine/2* e *printBoardBody/3*.



				*
*				
			N	

Figura - Visualização do puzzle antes de aplicar o predicado solver

O			*	*
*	*	O		
*	M	*		
	*		N	*
		*	*	O

Figura - Visualização do puzzle depois de aplicar o predicado solver

5 Resultados

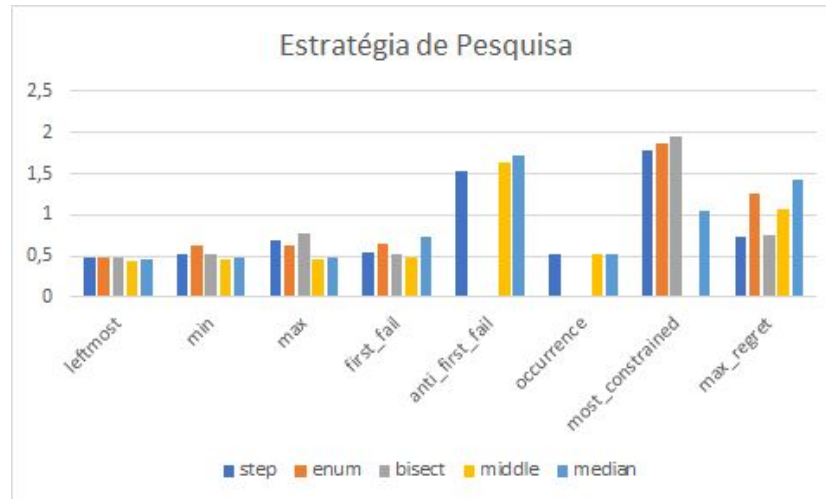
Os resultados avaliados dividem-se em duas partes:

1. Análise de um puzzle que varia consoante o N. Foi medido o tempo de execução, número de retrocessos e número de restrições aplicadas.
2. Análise de um puzzle constante, com N=40, variando-se apenas as opções que se colocava no *labeling*. Foi apenas medido o tempo de execução.

Para a primeira análise foi gerado um puzzle aleatório, 6 vezes. O seguinte gráfico demonstra os resultados calculados através das média das 6 medições. É importante frisar que os puzzles de N igual a 5,6 e 7 são puzzles pré-calculados (i.e é feito o random dos puzzles demonstrados no site do jogo) e por isso apresentam maior número de retrocessos que os outros puzzles que iniciam com uma lista vazia.

Puzzle (N)	Tempo(s)	Retrocessos	Constraints
5	0,01	0,5	302
6	0,01	1	449
7	0,025	2	662
25	0,04	0	11800
50	0,36	0	48600
100	14,45	0	197200

Para a segunda análise foi escolhido um puzzle de $N=40$. As linhas da tabela apresentada representam de que modo as escolhas são feitas para cada variável selecionada e as colunas são as opções que controlam a ordem de escolha da próxima variável. É importante frisar que as barras que não aparecem para determinada estratégia de pesquisa que determina a ordem da próxima variável (colunas) significa que o tempo dessa pesquisa é de uma a ordem de grandeza muito superior às restantes e por isso foi considerada irrelevante.



Nota: Na parte dos anexos é possível ver todos os gráficos e tabelas calculados para esta secção

6 Conclusão e Trabalho Futuro

O projeto teve como principal objetivo aprender o conteúdo lecionado nas aulas teóricas e pôr em prática esse conhecimento. Conclui-se que a aplicação de restrições ao invés do método mais básico, ensinado primeiro, traz imensos benefícios em termos de otimização, eficiência, legibilidade e organização do código escrito, para certos problemas.

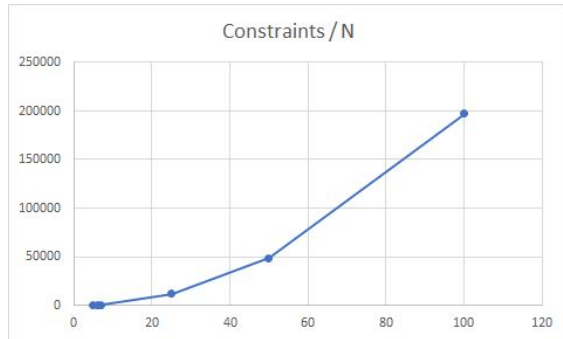
Ao longo do desenvolvimento deste projeto foram encontradas algumas dificuldades nomeadamente a escolha de predicados e implementação das restrições. Inicialmente, implementou-se restrições de várias linhas, complexas e desnecessárias. Ao usar os predicados *automaton* e *transpose* estas dificuldades foram ultrapassadas.

É de se notar que, existem alguns aspetos passíveis de serem melhorados, tais como, garantir uma solução única ou gerar um board de forma aleatória de forma mais eficiente. Apesar disso, o projeto apresentado é muito satisfatório pelos resultados apresentados acima.

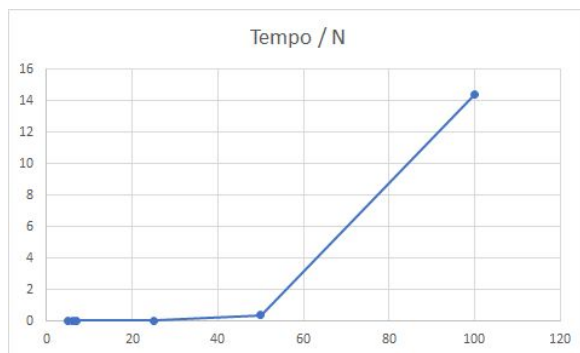
Concluindo, o projeto foi desenvolvido com muito sucesso, cumprindo-se todas as etapas pedidas. Além disso, contribuiu para o nosso enriquecimento em termos da unidade curricular de programação em lógica e suas derivadas.

7 Anexos

7.1 Tabelas e Gráficos



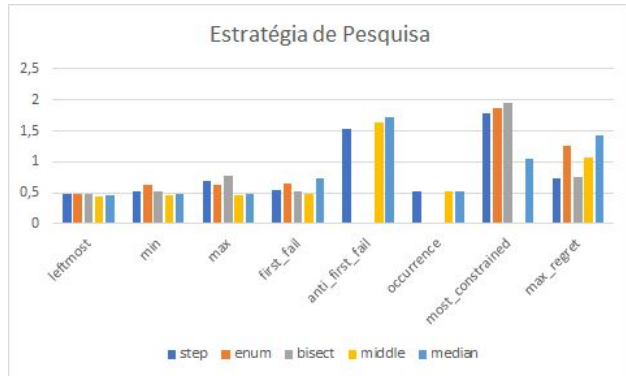
Variação das restrições aplicadas em função do tamanho do puzzle.



Variação da duração de resolução em função do tamanho do puzzle.

Puzzle (N)	Tempo(s)	Retrocessos	Constraints
5	0,01	0,5	302
6	0,01	1	449
7	0,025	2	662
25	0,04	0	11800
50	0,36	0	48600
100	14,45	0	197200

Variação do tempo de execução, número de retrocessos e de restrições para puzzles variáveis.



Estratégia de Pesquisa

7.2 Código fonte

lib.pl

```
:-use_module(library(clpfd)).
:-use_module(library(lists)).
:-use_module(library(system)).
:-use_module(library(random)).
:-use_module(library(between)).
:-[board].

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,
    fd_statistics.

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

```
:-[board].
:-[lib].
:-[generator].
:-[puzzles].

main(N, Matrix) :-
    % Generates a board based on predefined boards
    generate_board(N, Matrix),
    once(printBoard(Matrix, N)),
    % Solves the random puzzle
    solver(Matrix, N),
    % prints the game Board
    %once(printBoard(Matrix, N)).
```

```
solver(Matrix, N, Y) :-
    % Resets time for statistics
    reset_timer,
    % Stores the matrix as 1 dimension
    append(Matrix, OneListMatrix),
    % Specifies the domain of the matrix
    domain(OneListMatrix, 0, 4),
    % Restrictions for lines
    solveMatrix(Matrix, N),
    % Restrictions for columns
    transpose(Matrix, TMatrix),
    solveMatrix(TMatrix, N),
    % Labeling of the one list matrix
    labeling([], OneListMatrix),
    % prints elapsed time
    print_time.
```

```

% Solves received matrix (solves only the Lines)
solveMatrix([], _).
solveMatrix([H|T], N) :-
    automaton(H, _, H, [source(s), sink(s3), sink(o2)],
        [
            arc(s, 0, s),
            arc(s, 1, o3),
            arc(s, 4, s1),

            arc(s1, 0, s1, [C+1]),
            arc(s1, 2, s2),
            arc(s1, 3, s4),
            arc(s1, 4, o1),

            arc(s2, 0, s2, [C-1]),
            arc(s2, 0, s5),
            arc(s2, 4, s3),

            arc(o1, 0, o1),
            arc(o1, 1, o2),

            arc(o2, 0, o2),

            arc(o3, 0, o3),
            arc(o3, 4, o4),

            arc(o4, 0, o4),
            arc(o4, 4, o2),

            arc(s3, 0, s3),

            arc(s4, 0, s4, [C-1]),
            arc(s4, 4, s3)
        ],
        [C], [0], [DeltaDist]
    ),
    Naux is N - 1,
    solveLine(H, DeltaDist, 0, Naux),
    solveMatrix(T, N).

```

```

% Restricts each line (2 dots and 1 letter per line + restricts line)
solveLine([], _, _, _).
solveLine([H | T], DeltaDist, Counter, Length):-
    (
        (Counter \= 0, Counter \= Length) ->
            fd_batch([DeltaDist #= 0 #<= H #= 2, DeltaDist #\= 0 #<= H #= 3]); true
    ),

    CounterAux is Counter + 1,
    solveLine(T, DeltaDist, CounterAux, Length).

```