**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**

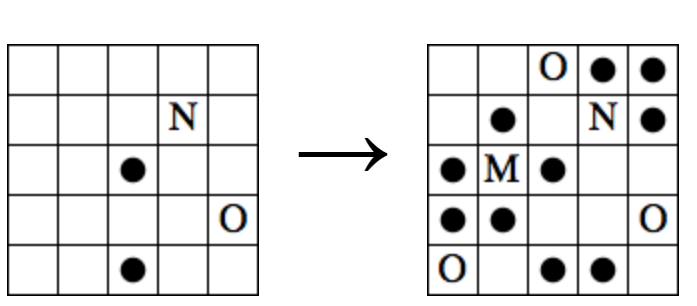
O projeto foi desenvolvido no âmbito da unidade curricular de Programação em Lógica de 3º ano do curso Mestrado Integrado em Engenharia Informática e de Computação. Para tal, foi necessário implementar uma possível resolução para um problema de um puzzle, com restrições sendo que optou-se pelo jogo chamado 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 do problema e a sua implementação utilizando o SICStus Prolog.
  + **Função de Avaliação**: descrição da forma de avaliar a solução obtida e a 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**: conclusões retiradas deste projeto, resultados obtidos, como o trabalho poderia ser melhorado.
* **Bibliografia**: Livros, artigos, páginas Web, usados para desenvolver o trabalho.
* **Anexos**: Código fonte, ficheiros de dados e resultados

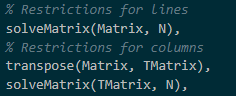
**2. Descrição do problema**

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 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.

**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 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 optou-se por dividir o problema em dois. Em primeiro lugar, recebendo o tabuleiro como uma lista de listas faz-se as restrições para cada linha do puzzle. Em seguida, com auxílio ao predicado *transpose/2*, é calculada a transposta da matriz recebida e chama-se o predicado que restringe as linhas, sendo que, desta vez restringe as colunas.

Para isso, elaborou-se 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).

Figura 2 – Predicado *solveMatrix/2*

**3.1 Variáveis de decisão**

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

Figura 3 – Aplicação do domínio

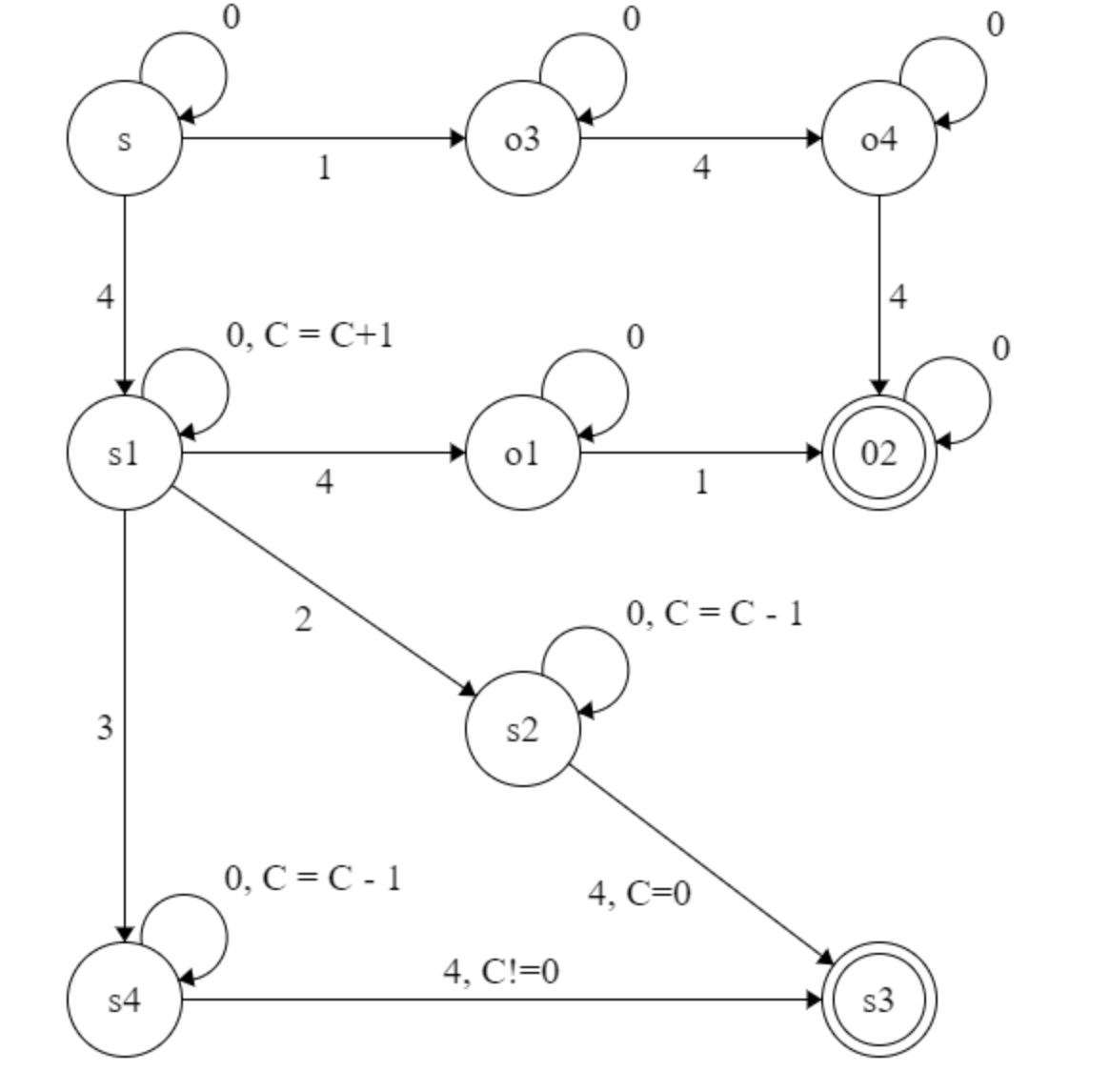
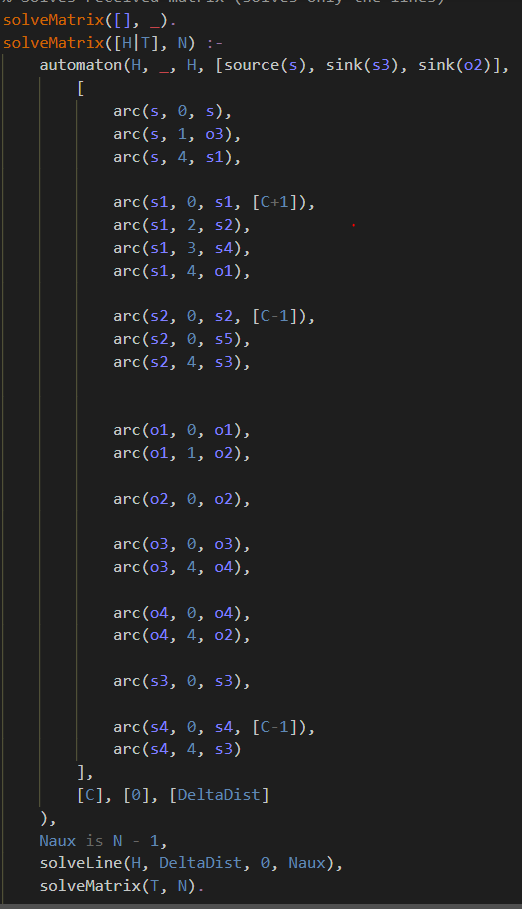
**3.2 Restrições**

Figura 5 – Representação gráfica do automâto

Figura 4 – Predicado *solveMatrix/2* que contém o automâto



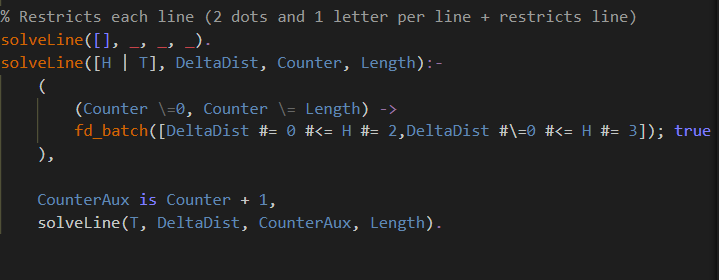
Todas as restrições impostas são implementadas com o uso de um automata (NFA) representado na figura acima. É responsável por garantir que por cada linha e coluna hajam 2 pontos e 1 letra. Além disso, garante que as letras M e N estão entre 2 pontos e a letra O está fora dos pontos (i.e não está entre 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≠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 representa o único contador do automata. Este contador é a variação das distâncias entre os 2 pontos e a letra, ou seja, se C=0 trata-se da letra M e se C≠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 quer dizer que o contador do autômato tem quer igual a zero, isto é, a distância dos pontos à 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 à letra N é diferente.

Figura 6 – Predicado *solveLine/4* que impõe restrições para o contador do automâto



**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 no programa se 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** é o número de vezes que uma restrição foi reatada, **entailments** é o número de vezes que um (‘dis’) entailment foi detectado, **prunings** é o número de vezes que um domínio foi reduzido, **backtracks** é o número de vezes que foi encontrada uma contradição por um domínio ter ficado vazio ou uma restrição global ter falhado e finalmente **constraints** é 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\_timer/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 do bocado de código acabar, chamar o predicado*print\_time/0* para imprimir o tempo decorrido.

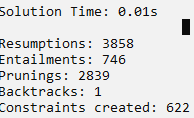
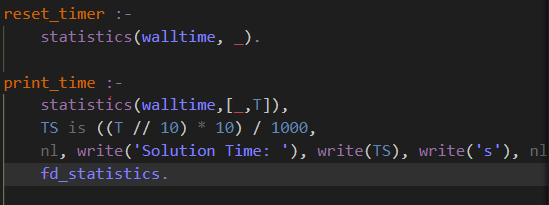
Para imprimir as estatísticas de execução específicas do solver clp(fd) é apenas preciso fazer a chamada ao predicado *fd\_statistics/0* que é chamado dentro e no final do predicado *print\_time/0*já que sempre que se quer imprimir o tempo também se quer ver as estatísticas até aquele momento.

Figura 8 – Exemplo da visualização na consola, das estatísticas e do tempo

Figura 7 – Predicados *print\_time/0* e *reset\_timer/0*



**3.4 Estratégia de Pesquisa**

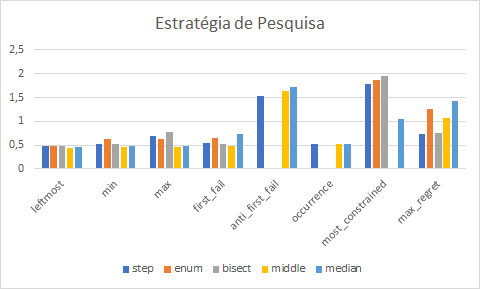
Foram testadas várias opções de pesquisa através dos valores que se foram passando no primeiro parâmetro do predicado *labelling/2***.** Ao nível da ordenação das variáveis, no labelling, para se saber como selecionar a próxima variável, passou-se no primeiro argumento um dos seguintes valores: **leftmost, min, max, first\_fail, anti\_fist\_fail, occurence, most\_contrained, max\_regre**t. Ao nível da seleção de valores, no labelling, para saber como selecionar 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, um par de valores resultantes da combinação de um valor da parte da ordenação de variáveis e um valor da parte da seleção 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:.

Figura 9 –Gráfico de barras acerca da estratégia de pesquisa

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 seleção com 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 seleção com os valores de ordenação de variáveis **anti\_first\_fail, occurence, most\_contrained e max\_regret** conduzem 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**

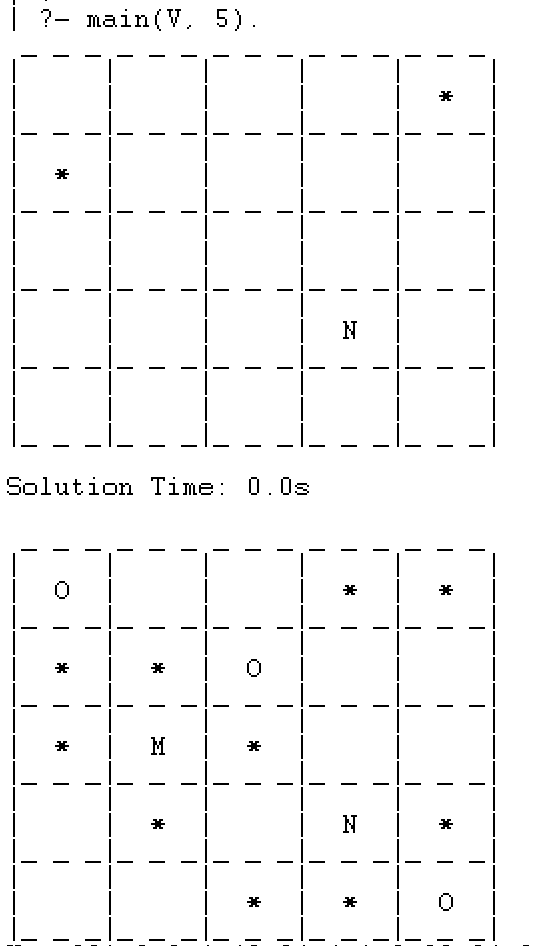
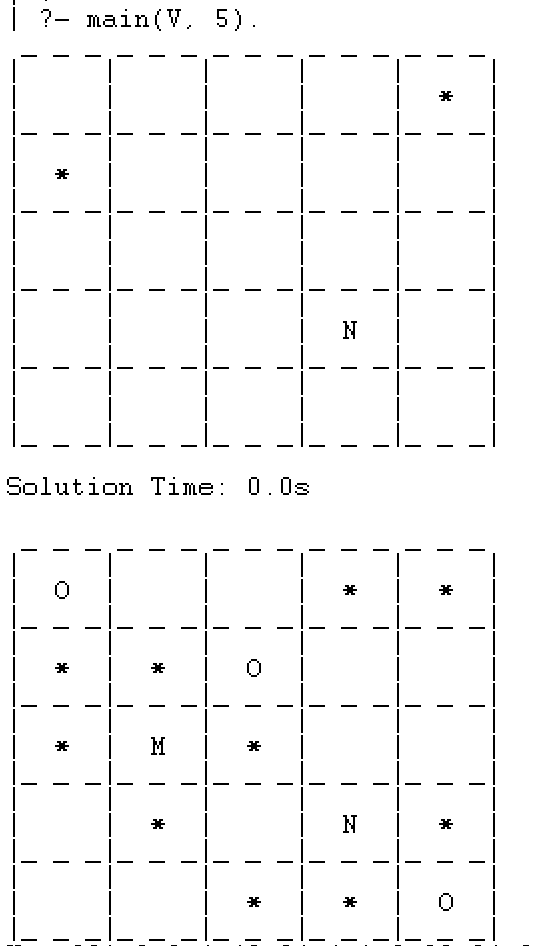
A visualização da solução é feita com recurso a predicados do ficheiro **board.pl** através dos quais imprime-se o tabuleiro de jogo inicialmente gerado e o puzzle resolvido. O predicado responsável pela apresentação do puzzle é o ***printBoard/2*** que recebe uma lista de listas e o tamanho de cada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

Figura 11 – tabuleiro 5x5 resolvido

Figura 10 – tabuleiro 5x5 por resolver



**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 colocam no *labeling.*  Foi apenas medido o tempo de execução.

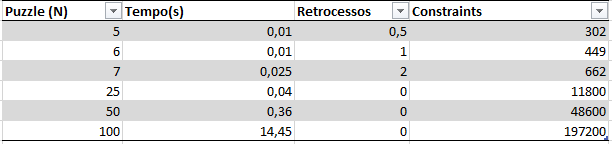
Para a primeira análise foi gerado um puzzle aleatório, 6 vezes. O seguinte gráfico demonstra os resultados calculados através das média das 6 medições. É importante frisar que os puzzles de N igual a 5,6 e 7 são puzzles pré-calculados (i.e é feito o random dos puzzles demonstrados no site do jogo) e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

Figura 11 – Tabela de Resultados dos tempos, retrocessos e Constraints (restrições)

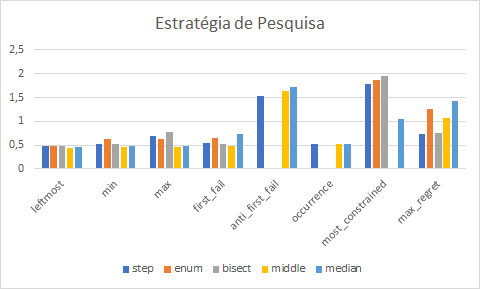
Para a segunda análise foi escolhido um puzzle de N=40. As linhas da seguinte tabela representam de que modo as escolhas são feitas para cada variável selecionada e as colunas são as opções que controlam a ordem de escolha da próxima variável. É importante frisar que as barras que não aparecem para determinada 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.

Figura 12 –Gráfico de barras acerca da estratégia de pesquisa

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.

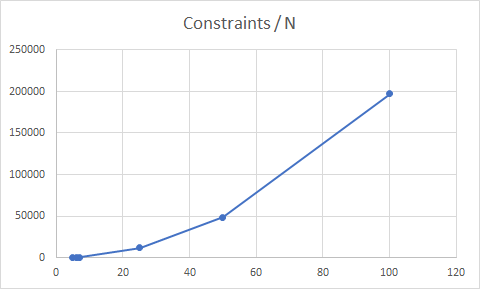
Ao longo do desenvolvimento deste projeto foram encontradas algumas dificuldades nomeadamente a escolha de predicados e implementação das restrições. Inicialmente, implementou-se restrições de várias linhas, complexas e desnecessárias. Ao usar os predicados *automaton* e *transpose* estas dificuldades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

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

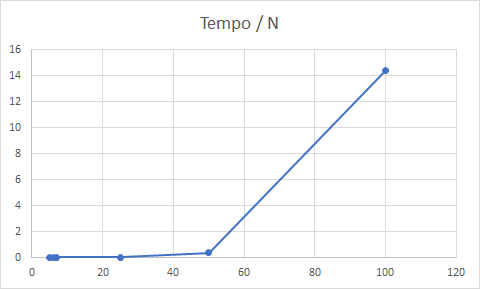
Concluindo, o projeto foi desenvolvido com muito sucesso, comprindo-se todas as etapas pedidas. Além disso, contribuiu para o enriquecimento em termos da unidade curricular de programação em lógica e suas derivadas.

**Anexos**

**7.1 Tabelas e Gráficos**

****

Variação das restrições aplicadas em função do tamanho do puzzle

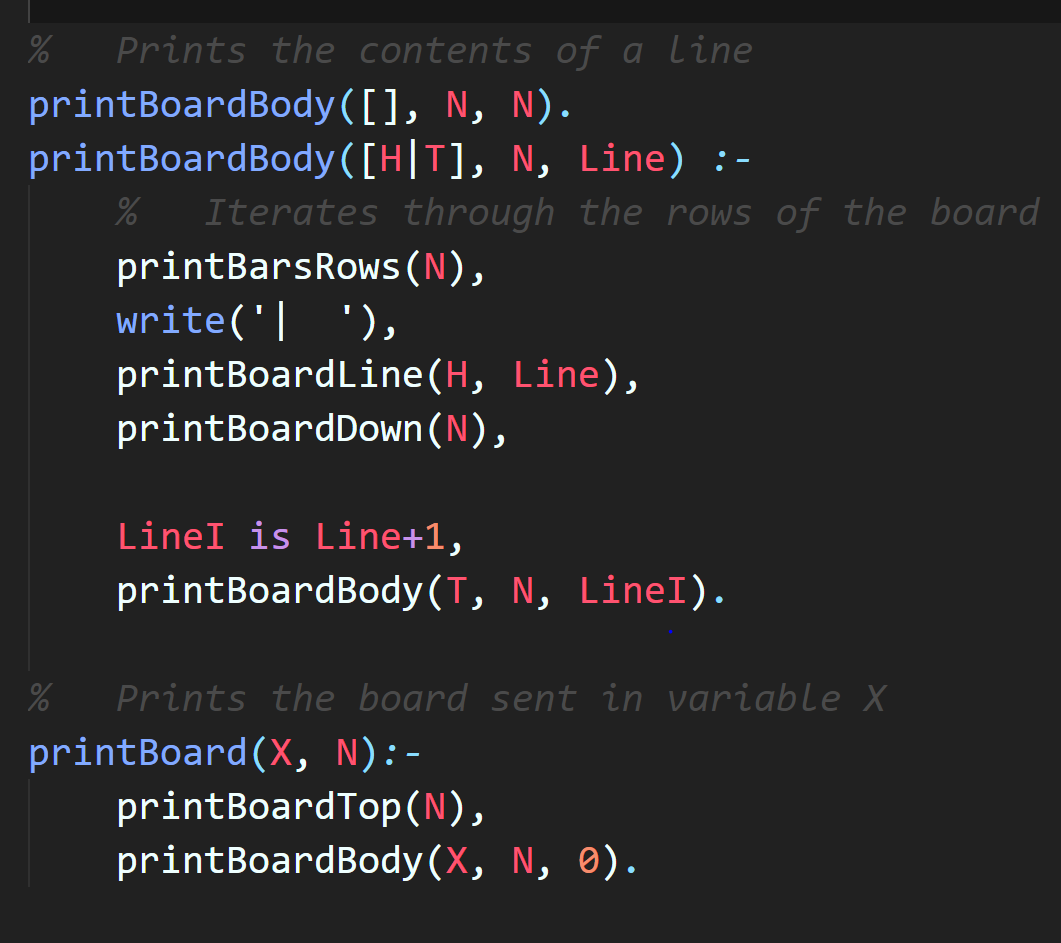
****

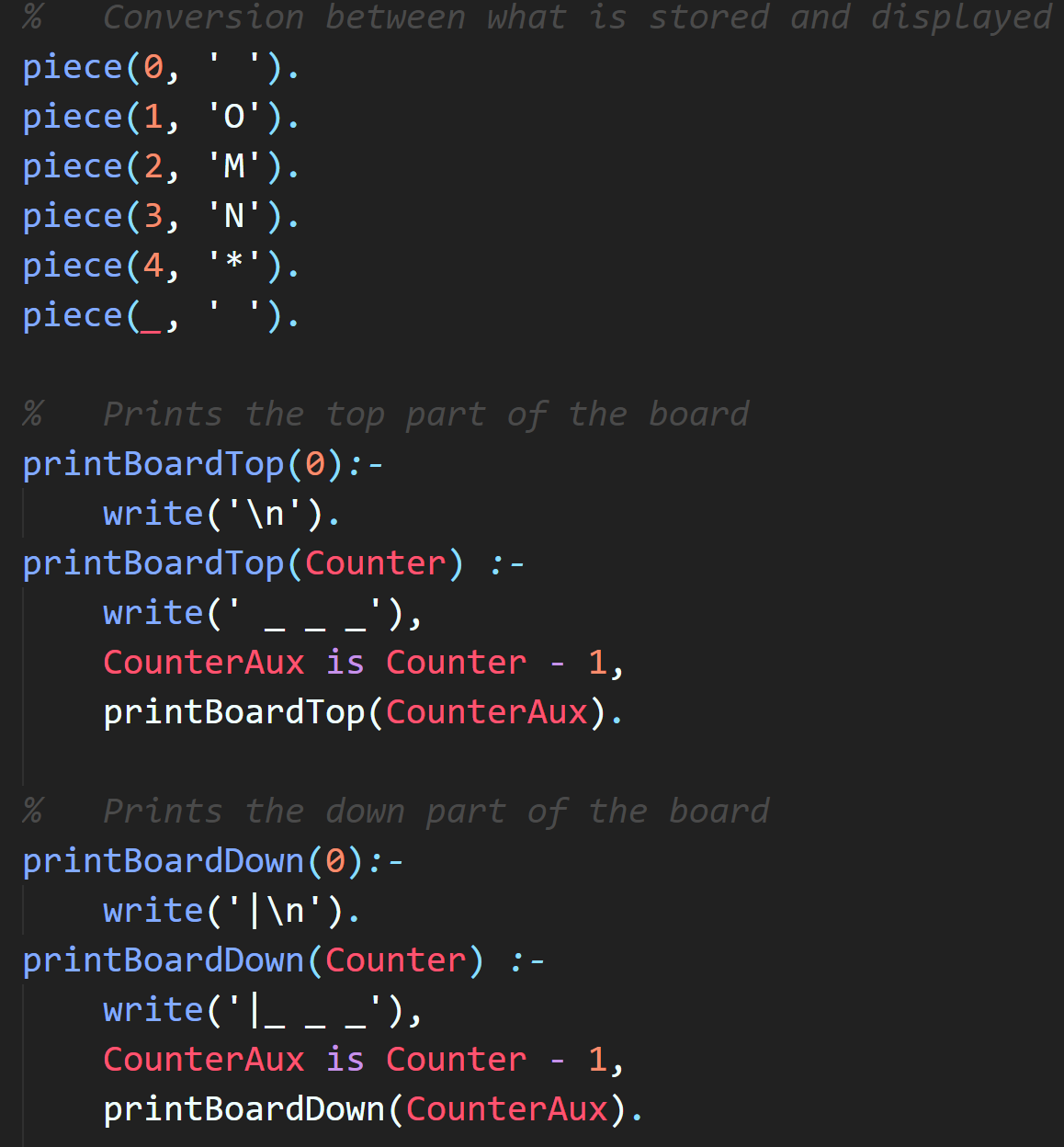
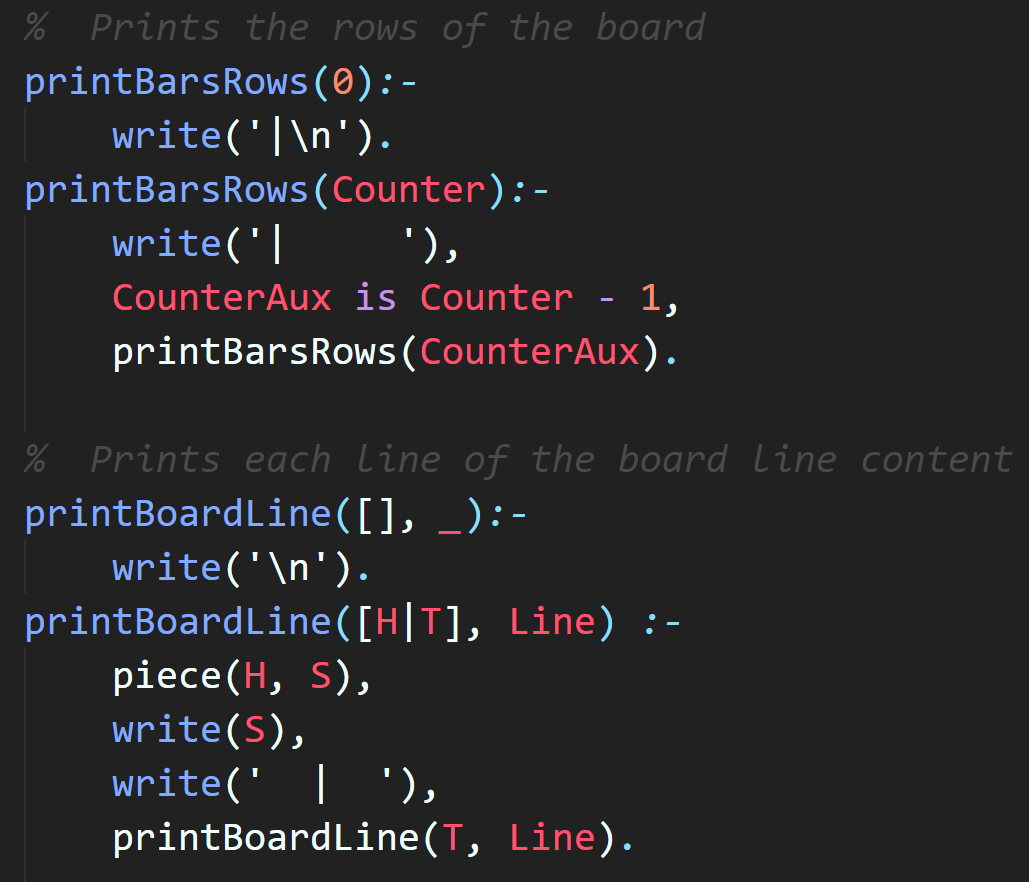
Variação do tempo de execução, número de retrocessos e de restrições para puzzles variáveis.

**7.2 Código fonte**

***lib.pl***



***board.pl***



**S*olver.pl***

