Programação em Lógica com Restrições: C-Note

Davide Castro e Rui Pinto

FEUP-PLOG, Turma 3MIEIC03, Grupo C-Note\_4

Faculdade de Engenharia da Universidade do Porto

Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal

**Resumo.** O projeto foi desenvolvido no âmbito da unidade curricular de Programação em Lógica, com o objetivo de resolver um problema de otimização ou decisão com Restrições, usando o SICStus Prolog como Sistema de desenvolvimento. Neste caso, foi escolhido o puzzle C-Note que consiste em completar uma tabela de forma a que a soma dos elementos das linhas e colunas seja sempre cem.

**Keywords.** FEUP, Prolog, Restrições, Lógica, SICStus.

1. Introdução

Este projeto foi desenvolvido tendo como base os conhecimentos adquiridos na unidade curricular de Programação em Lógica do 3º ano do Mestrado Integrado em Engenharia Informática e Computação.

Tendo como problema a resolução de um problema de decisão ou otimização com restrições, foi escolhido o problema “C-Note”, sendo necessária a resolução com o uso do SICStus Prolog e a sua biblioteca CLP(FD) de programação em lógica com restrições. Com este trabalho prático, foi possível ao grupo alargar os conhecimentos em relação às restrições em programação lógica e como aplicá-los de forma prática em problemas de otimização e decisão combinatória, assim como entender as suas vantagens e desvantagens em termos computacionais.

O problema escolhido consiste num puzzle simples, fazendo uso de uma grelha de valores dados a um jogador, que tem que completar com dígitos de forma a que a soma de todos os valores das colunas e linhas seja igual a cem.

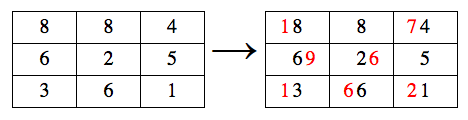
Este artigo segue, assim, a seguinte estrutura:

* **Descrição do Problema:** descrição detalhada do problema escolhido para o trabalho prático.
* **Abordagem:** descrição da abordagem do grupo em relação à solução do problema.
  + **Variáveis de Decisão:** descrição das variáveis de decisão usadas e os seus domínios.
  + **Restrições:** descrição das restrições necessárias ao problema e a sua implementação em programação lógica usando o SICStus Prolog.
* **Visualização da Solução:** explicação da implementação da visualização da solução em modo de texto e dos predicados usados para esse fim.
* **Experiências e Resultados**
  + **Análise Dimensional:** exemplos de execução do problema em várias instâncias com dimensões diferentes e análise dos resultados.
  + **Estratégias de Pesquisa:** demonstração e comparação dos resultados obtidos em testes ao problema usando diferentes estratégias de pesquisa.
* **Conclusões e Trabalho Futuro:** conclusões retiradas do trabalho prático, assim como discussão sobre os resultados obtidos, vantagens e desvantagens da solução e como poderia ser melhorada.

1. Descrição do problema

O problema escolhido, intitulado de “C-Note”, é um puzzle constituído por uma grelha que, inicialmente, contém um digito em cada célula, diferente de zero. O objetivo é completar a tabela da seguinte forma: deve ser colocado à direita ou à esquerda, mas nunca ambos, do digito de cada célula, um qualquer outro dígito, havendo a opção de não colocar nenhum, que equivale a colocar um zero à esquerda do dígito. Com a colocação do novo dígito é formado um valor novo em cada célula, por exemplo, se o valor inicial era 8, ao colocar um dígito 9 à esquerda, o valor da célula passa a ser noventa e oito (98).

No final, a soma de todos os valores em todas as linhas e todas as colunas da grelha, tem que ser igual a cem.



**Fig. 1.** Exemplo de problema e a sua solução

1. Abordagem

Neste trabalho prático, abordamos este problema de satisfação de restrições (PSR) desenvolvendo um solucionador com o uso a biblioteca clp(FD), de programação com restrições do SICStus Prolog. Para isto, implementamos as restrições necessárias em Prolog, tendo em consideração as variáveis de decisão a usar.

Para representar a grelha inicial foi usada uma lista de listas, que representam as linhas da grelha, com os valores ordenados pelo número da coluna.

* 1. Variáveis de Decisão

De forma a conseguir a solução para o problema, são usadas duas variáveis:

* Uma tabela representada por uma lista em que cada elemento é também uma lista que representa uma linha da grelha, contendo assim os seus valores por ordem. Esta tabela contém os dígitos a colocar em cada célula de forma a resolver o problema. É chamada no código de **DigitGrid**. //nao sei
* Uma tabela, representada por uma lista que resulta da junção de todas as linhas, contendo os valores finais da solução, constituídos pelo valor correspondente em cada célula na DigitGrid, junto com o valor inicial da célula no problema, formando os valores da solução (**ResultGrid**).
  1. Restrições

De forma a seguir as regras do puzzle e solucioná-lo de forma eficiente, foram aplicadas as seguintes restrições:

* **Os valores a inserir são números entre 0 e 9**.

Sendo S um dígito a inserir numa célula:

S in 0..9

* **Os valores resultantes da solução em cada célula são números de 1 a 99**.

Sendo R um valor da grelha de resultado:

R in 1..99

* **A soma dos valores em cada linha e coluna é igual a 100**.

Sendo Line e Column, uma linha e coluna da grelha resultante, respetivamente:

sum(Line, #=, 100)

sum(Column, #=, 100)

Estas restrições são aplicadas a todas as linhas e colunas.

* **O valor inserido está à direita ou à esquerda do dígito inicial**

Quando o dígito inserido está à esquerda, o valor final será o valor inserido multiplicado por 10 mais o valor inicial. Pelo contrário, se o valor inserido está à direita, o valor final será o valor inicial multiplicado por 10 mais o valor inserido.

Ou seja, para cada célula, sendo R o valor final, H o valor inicial e S o valor inserido:

R #= H\*10 + S #\/ R #= S\*10 + H

1. Visualização da Solução

Para visualizar a solução do problema em modo de texto foi implementado um predicado **presentResult(**Original, Result, N, M, OriginalM). Este recebe a lista contendo os valores iniciais do problema (Original), a lista contendo a solução (Result), e as dimensões do problema (N, M e OriginalM).

presentResult([], [], 0, \_, \_).

presentResult([Original | RestOriginal], [First | Rest], N, M, OriginalM) :- % Input | Result

    write('|'),

    RigthDigit is mod(First, 10),

    presentNum(Original, RigthDigit, First),

    NewM is M - 1,

    decrementN(N, NewM, NewN, NewMAgain, OriginalM),

    presentResult(RestOriginal, Rest, NewN, NewMAgain, OriginalM).

No predicado de visualização são percorridas as listas Original e Result, usando o predicado **presentNum(**Original, RightDigit, Sol), sendo Original o valor inicial na célula, RightDigit o modulo da divisão do valor na solução por 10 e Sol o valor na solução.

Perante este predicado existem três situações possíveis:

* + - Se Sol for menor que 10, ou seja, só contém um dígito, significa que foi colocado um zero à esquerda do valor inicial, que equivale a não inserir nenhum dígito, o número é impresso no centro da célula.

presentNum(\_, \_, Number) :-

     Number < 10,

     write('  '), write(Number), write('  |').

* + - Se RightDigit for igual ao dígito inicial, pode-se concluir que o novo dígito foi colocado à esquerda, logo o valor resultante é inserido na célula com mais espaço à direita.

presentNum(Original, Original, Number) :-

     write(' '), write(Number), write('  |').

* + - Se nenhuma das outras situações se confirmou, significa que o dígito foi inserido à direita, pelo que a solução é impressa com um deslocamento para a direita.

presentNum(\_, \_, Number) :-

     write('  '), write(Number), write(' |').

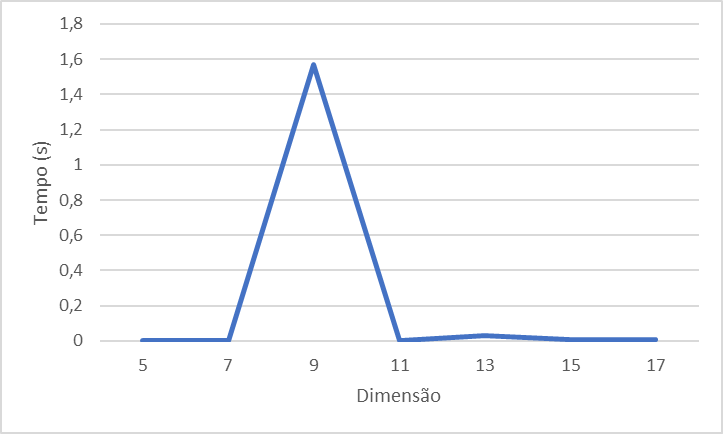
1. Experiências e Resultados

Para testar a solução implementada para o problema, foram efetuadas várias experiências, tendo em conta a dimensão do problema e diferentes estratégias de pesquisa da solução.

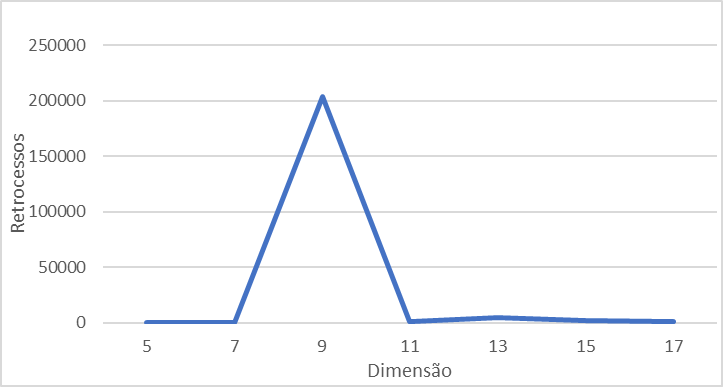
* 1. Análise Dimensional

Para este problema, é considerado que a sua dimensão é igual ao tamanho das linhas e colunas, ou seja, a dimensão da grelha do puzzle. Por isso, executámos o solucionador implementado com várias instâncias do problema mas com diferentes dimensões para comparar os resultados obtidos.

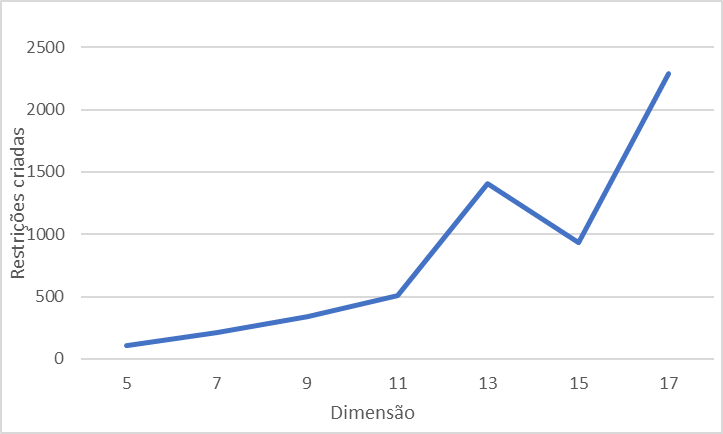
Pode-se observar alguns dos resultados nos gráficos seguintes:



**Fig. 2.** Tempo de labeling em relação à dimensão



**Fig. 3.** Número de retrocessos em função da dimensão

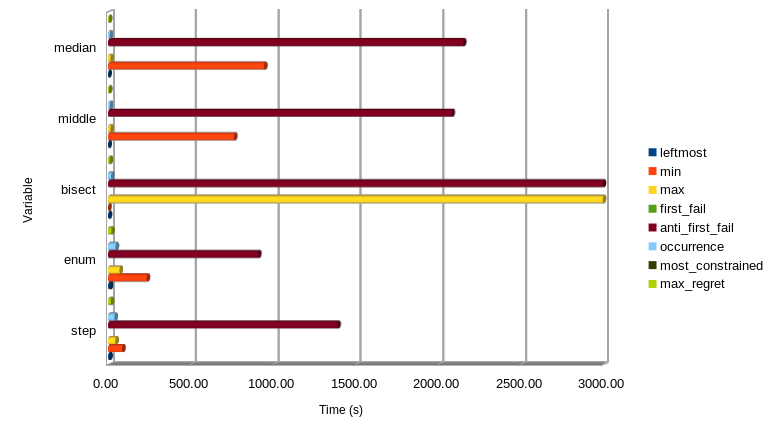


**Fig. 4.** Número de restrições criadas em função da dimensão

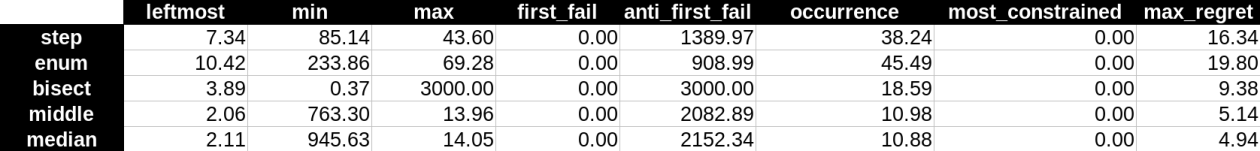
Com estas experiências, conseguimos concluir que, com o aumento da dimensão, o número de restrições tende a crescer (figura 4), devido ao maior número de linhas e colunas a que vão ser aplicadas.

No entanto, no tempo e número de retrocessos, isso não será sempre observado, como é possível ver nos gráficos das figuras 2 e 3. Quanto a isto, o grupo concluiu que a performance do solucionador depende dos dígitos iniciais presentes na grelha do puzzle. Havendo muitas combinações de valores diferentes possíveis para cada dimensão, e considerando que para cada dimensão foram testadas grelhas iniciais diferentes, a performance, tal como os retrocessos, pode variar bastante de experiência para experiência, não havendo assim uma clara evolução da eficiência do programa com o variar da dimensão.

* 1. Estratégias de Pesquisa



**Fig. 5.** Gráfico do tempo de labelling nas várias combinações



**Fig. 6.** Tabela de tempos de labeling nas várias combinações

As experiências foram realizadas usando o input com a grelha 5x5

[[3,1,4,2,5],[3,1,4,2,5] ,[3,1,4,2,5] ,[3,1,4,2,5] ,[3,1,4,2,5]].

Das diferentes estratégias de pesquisa testadas, o grupo conseguiu concluir que as mais eficientes neste programa foram o *first fail* e *most constrained*, com a qual foi possível executar o solucionador com tempos de execução mínimos. Pelo contrário, usando *anti first fail* foram obtidos os piores resultados, com tempos muito superiores aos observados com as restantes estratégias.

Devido a estes resultados, o grupo optou por usar as opções de *first fail* e *most constrained* no labeling, visto que permitem obter a solução de maneira muito mais eficiente.

1. Conclusões e Trabalho Futuro

Em suma, com este trabalho prático, o grupo conseguiu implementar com sucesso um *solver* para um problema de satisfação de restrições, neste caso, o puzzle “C-Note”, possibilitando maior conhecimento sobre a utilização de programação em lógica com restrições em linguagem Prolog.

Ao longo do desenvolvimento, foi reforçada a importância da estruturação dos problemas e do estudo de diferentes técnicas e estratégias para alcançar as soluções.

Foi também destacada a possível aplicabilidade da programação com restrições em problemas futuros e as suas vantagens em relação a outros métodos de resolução de problemas de decisão e otimização.