Resolução de um Problema de Decisão usando Programação em Lógica com Restrições: C-Note

Miguel Silva e Rodrigo Abrantes

FEUP-PLOG, Turma 3MIEIC04, Grupo C-Note_2

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

Resumo. Este projeto foi desenvolvido utilizando o Sistema de Desenvolvimento SICStus Prolog no âmbito da unidade curricular de Programação em Lógica [1], tendo como objetivo resolver um problema de decisão/otimização através do uso de restrições. O problema escolhido é o puzzle *C-Note* [2], onde nos é dada uma grelha com um dígito em cada célula e tem como finalidade obter uma nova grelha em que a soma dos números de cada linha ou coluna é igual a 100, isto adicionando novos dígitos à esquerda ou à direita do dígito que já se encontra na célula. Assim, utilizando a linguagem de *Prolog* e a biblioteca *clpfd* [3] foi possível a resolução deste problema.

Keywords: *C-Note*, Problema de Decisão/Otimização, *SICStus*, *Prolog*, FEUP

1 Introdução

O projeto foi desenvolvido no âmbito da unidade curricular de Programação em Lógica do curso Mestrado Integrado em Engenharia Informática e de Computação. Para tal, foi necessário apresentar uma solução para um problema de decisão ou otimização utilizando Prolog com Restrições. O problema escolhido trata-se de um problema de decisão, o *C-Note***.**

O artigo possui a seguinte estrutura:

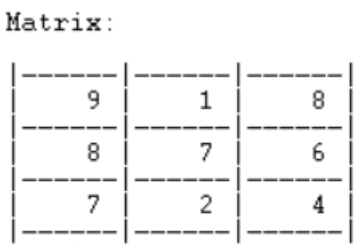
- **Descrição do Problema:** descrição detalhada do problema de otimização escolhido, incluindo todas as restrições envolvidas.
- **Abordagem:** descrição da modelação do problema como um PSR.
  - **Variáveis de Decisão:** descrição das variáveis de decisão e dos seus respetivos domínios,
  - **Restrições:** descrição das restrições rígidas e flexíveis do problema e a sua implementação utilizando o *SICStus Prolog*.
- **Visualização da Solução:** Explicação dos predicados que permitem visualizar a solução em modo de texto.
- **Experiências e Resultados:**
  - **Análise Dimensional:** Inclui diferentes exemplos de execução em instâncias do problema com diferentes dimensões e análise os resultados obtidos.
  - **Estratégias de Pesquisa:** demonstração do teste de diferentes estratégias de pesquisa (heurísticas de escolha de variável e de valor), comparando os resultados obtidos.
- **Conclusões e Trabalho Futuro:** conclusões retiradas deste projeto, resultados obtidos, vantagens e limitações da solução proposta e possíveis aspetos a melhorar.
- **Referências**: fontes bibliográficas usadas, incluindo livros, artigos, páginas Web, entre outros.
- **Anexo:** código fonte, ficheiros de dados, resultados detalhados, entre outros.

2 Descrição do Problema

O puzzle escolhido, *C-Note*, trata-se de uma grelha quadrada de tamanho arbitrário, contendo, em cada célula, um dígito. O objetivo do puzzle é adicionar dígitos antes ou depois do dígito já presente na célula fornecida de forma a que a soma de cada linha e coluna da grelha seja igual a 100.

3 Abordagem

Para a resolução deste problema foi utilizado, para representar a grelha inicial, uma lista única da grelha, seguida do número de células por linha. Por exemplo, o input [9,1,8,8,7,6,7,2,4], 3 representa a seguinte grelha



3.1 Variáveis de Decisão

Para a resolução do problema, é inicializada uma lista de váriaveis com o tamanho da grelha inicial, variável *Output*, ou seja, para o caso acima, esta lista teria tamanho de 9. No fim da aplicação de restrições através das funções *constrainSum* e *addRestrictioms* e da atribuição de valores às variáveis através do *labeling* esta lista irá constituir a solução ao puzzle. Assim, a váriavel *Output* teria, no final, os valores [19,1,80,8,76,16,73,23,4].

3.2 Restrições

Para o problema escolhido existem basicamente duas restrições que terão de ser impostas para o cálculo da solução.
A soma das linhas e das colunas tem que ser igual a Value (Valor dado pelo utilizador aquando da chamada do predicado para a resolução do problema, com valor normalmente igual a 100), sendo utilizado o predicado *constrainSum* para a aplicação desta restrição.
Os dígitos nas células iniciais terão de estar presentes na solução obtida, sendo então utilizado o predicado *addRestrictions* para garantir que isto acontece.

4 Visualização da Solução

Por forma a correr o programa deverá ser usado o predicado *cnote(+Board, +No, +Value)*, sendo *Board* a grelha inicial, *No* o número de células por linha e *Value* o valor para a soma.
Para uma melhor demonstração da solução obtida pelo programa, após ser obtida esta, é chamado o predicado *printBoard(+Board)* que apresenta de forma mais *user friendly* esta mesma solução, sendo usados no total 5 predicados.
O predicado principal, *print_board(+Board, +Message)*, começa por calcular o número de elementos de uma linha. De seguida, mostra a mensagem que lhe é passada, sendo esta utilizada para fazer a distinção entre uma grelha-problema e uma grelha-solução. É ainda chamado o predicado *print_separation(+Counter, +RowLength)*, que imprime o limite exterior da grelha.
Inicialmente, o predicado *printBoard* imprime um Separador inicial e chama o predicado *printMatrix*, o qual irá dar display da matriz em si.

```
printBoard(Board):-
    nl,
    nth1(1, Board, Row),
    length(Row, Length),
    write('|'),
    print_separator(0, Length),
    nl,
    printMatrix(Board, 0, Length).
```

Posteriormente, no predicado *printMatrix*, são chamados os predicados *printLine* e *printSeparator*, sendo que *printLine* chama, para cada elemento da linha, o predicado *printElement*.

```
%prints the current matrix in a grid-like way
printMatrix([], _, _).
printMatrix([H|T], X, Length):-
    write('|'),
    X1 is X+1,
    printLine(H),
    nl,
    write('|'),
    print_separator(0, Length),
    nl,
    printMatrix(T, X1, Length).
```

```
print_separator(Length, Length).
print_separator(Aux, Length):-
    write('------|'),
    NewAux is Aux+1,
    print_separator(NewAux, Length).
```

```
%prints a matrixs line
printLine([]).
printLine([H|T]):-
    printElement(H),
    printLine(T).

printElement(H):-
    H < 10,
    write('    '),
    write(H),
    write(' |').
printElement(H):-
    H >= 10,
    H < 100,
    write('   '),
    write(H),
    write(' |').
printElement(H):-
    H >= 100,
    H < 1000,
    write('  '),
    write(H),
    write(' |').
printElement(H):-
    H >= 1000,
    write(' '),
    write(H),
    write(' |').
```

Assim, a utilização destes predicados permite imprimir na consola do *SICStus* a solução desta forma:

```
Matrix:

|------|------|------|
|    9 |    1 |    8 |
|------|------|------|
|    8 |    7 |    6 |
|------|------|------|
|    7 |    2 |    4 |
|------|------|------|
Result:

|------|------|------|
|   19 |    1 |   80 |
|------|------|------|
|    8 |   76 |   16 |
|------|------|------|
|   73 |   23 |    4 |
|------|------|------|
```

5 Experiências e Resultados

De modo a retirar conclusões dos resultados obtidos foram medidos o tempo de resolução, o número de retrocessos e o número de restrições criadas para cada solução.

5.1 Análise Dimensional

Seguem-se as condições de teste e as respetivas conclusões.

- **Fez-se variar o valor da soma das linhas e colunas, mantendo-se a dimensão da grelha (Tabela 2, Figura 1, Figura 2, Figura 3 em Anexo):** Através da análise dos gráficos obtidos verifica-se que o tempo de resolução do problema e o número de retrocessos variam de forma exponencial com o aumento do valor da soma. No entanto, o número de restrições criadas parece variar de forma linear consoante este parâmetro. Assim, pode-se concluir que o tempo de resolução depende do número de retrocessos e não do número de restrições criadas quando se aumento o valor da soma.
- **Fez-se variar o tamanho da grelha, mantendo-se constante a soma das linhas e colunas (Tabela 3, Figura 4, Figura 5, Figura 6 em Anexo):** Através da análise dos gráficos é possível verificar que o tempo de resolução do problema, o número de retrocessos e o número de restrições criadas variam exponencialmente com o aumento do tamanho da grelha.

5.2 Estratégias de Pesquisa

Foram também testadas várias estratégias de pesquisa por forma a comparar os resultados obtidos. Assim, foi utilizada a seguinte grelha com o valor da soma a 1000.

Matrix:

9	1	8
8	7	6
7	2	4

Os resultados obtidos com estes testes estão presentes na tabela 1, pelo que se pode comcluir que a melhor estratégia para o problema dado é utilizando a combinação *step* e *max_regret*, e a estratégia com o pior desempenho é a combinação *median* e *anti_first_fail*.

6 Conclusões e Trabalho Futuro

Este projeto teve como principal objetivo desenvolver um programa onde fossem aplicados conhecimentos acerca de Programação em Lógica com Restrições adquiridos nas aulas teóricas e práticas, por forma a resolver problemas de decisão/otimização em *Prolog*.

Ao longo do desenvolvimento do projeto fomos encontrando algumas dificuldades, maioritariamente relativas à aplicação e implementação de restrições, as quais foram superadas com a ajuda da biblioteca *clpfd* e dos slides fornecidos pelos docentes [4].

Nota-se também que poderia ter sido escolhido um método mais otimizado para a resolução deste problema para grelhas de maiores dimensões, o que pode ser visto pelas experiências realizadas.

Em suma, o projeto foi completado com sucesso visto que resolve o problema dado de forma correta e o seu desenvolvimento contribui para a nossa melhor compreensão do funcionamento das restrições em Programação em Lógica.

7 Referências

[1] sigarra.up.pt. (2020). FEUP - Programação em Lógica. [online] Available at: https://sigarra.up.pt/feup/pt/ucurr_geral.ficha_uc_view?pv_ocorrencia_id=459482 [Accessed 2 Jan. 2021].
[2] Friedman, E. (2011). C-Note Puzzles. [online] Available at: https://erich-friedman.github.io/puzzle/100/ [Accessed 28 Dec. 2020].
[3] sicstus.sics.se. (2020). SICStus Prolog: lib-clpfd. [online] Available at: https://sicstus.sics.se/sicstus/docs/4.1.0/html/sicstus/lib_002dclpfd.html [Accessed 4 Jan. 2021].
[4] moodle.up.pt (2020). Moodle UC: Programação em Lógica. [online] Available at: https://moodle.up.pt/course/view.php?id=1476 [Accessed 3 Jan. 2021].

8 Anexos

8.1 Tabelas e Gráficos

Tabela 1. Teste de Estratégia de Pesquisa

Grid: [9,1,8,8,7,6,7,2,4] Soma: 1000	leftmost	min	max	first_fail	anti_first_fail	occurrence	most_constrained	max_regret
step	1.96	13.03	3.41	7.13	9.98	1.83	7.44	1.78
enum	5.37	21.18	6.53	15.88	6.09	5.6	15.47	5.29
bisect	1.89	28.43	19.66	4.53	60.54	1.92	4.47	1.83
middle	6.31	24.23	6.87	16.09	83.41	6.16	17.56	6.13
median	6.36	20.74	7.14	16.26	93.34	6.27	17.18	5.93

Tabela 2. Variação da duração, retrocessos e restrições criadas em função da soma das linhas e colunas

Grid Dada	[9,1,8,8,7,6,7,2,4]		
Valor da Soma	Tempo (s)	Retrocessos	Restrições Criadas
100	0.03	153	187
200	0.06	866	519
250	0.06	1492	574
500	0.23	8812	744
750	0.65	21463	1043
1000	1.8	59602	1908
2500	39,77	935034	3610
5000	594.15	9105235	4219

Tabela 3. Variação da duração, retrocessos e restrições criadas em função da dimensão da grelha

Valor da Soma	100		
Grid Dada	Tempo (s)	Retrocessos	Restrições Criadas
[4,5,3,7] 2x2	0.02	11	27
[1,1,6,2,7,9,3,2,4] 3x3	0.03	368	350
[3,2,8,4,2,6,5,4,1,6,4,4,3,4,1,2] 4x4	0.34	40246	6695
[9,1,1,1,1,1,6,3,2,1,1,2,1,7,7,1,3,4,7,1,1,9,1,6,3] 5x5	240.3	36084851	587662

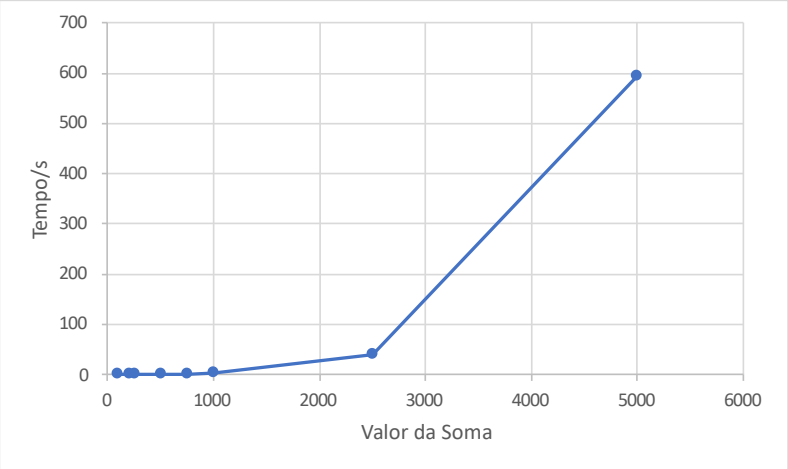


Fig. 1. Variação da duração de resolução em função da soma das linhas e colunas

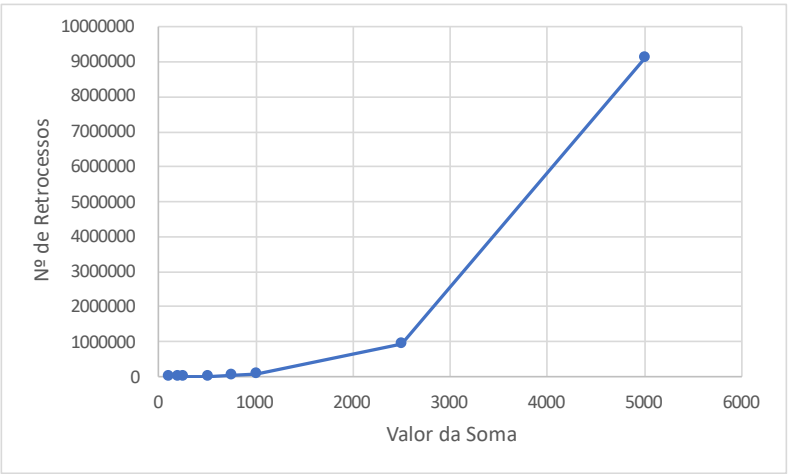


Fig. 2. Variação do número de retrocessos em função da soma das linhas e colunas

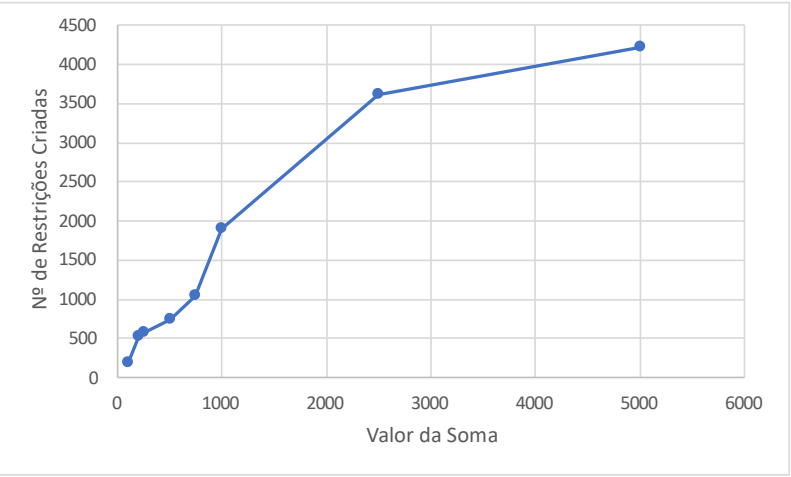


Fig. 3. Variação do número de restrições criadas em função da soma das linhas e colunas

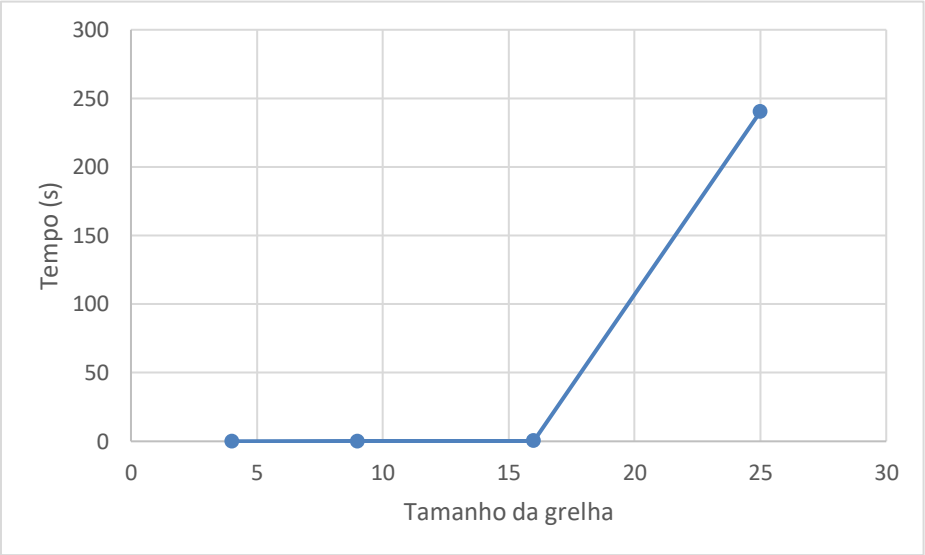


Fig. 4. Variação da duração de resolução em função da dimensão da grelha/tabuleiro

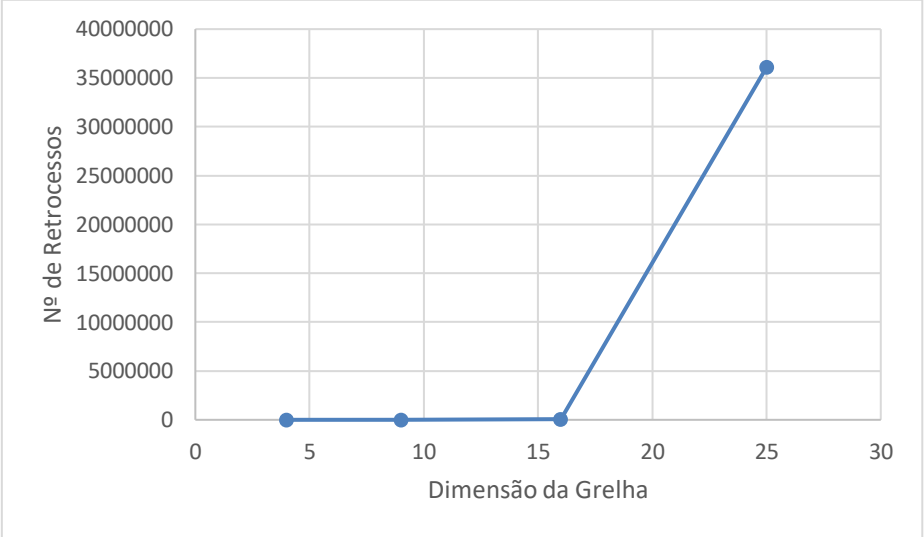


Fig. 5. Variação do número de retrocessos em função da dimensão da grelha/tabuleiro

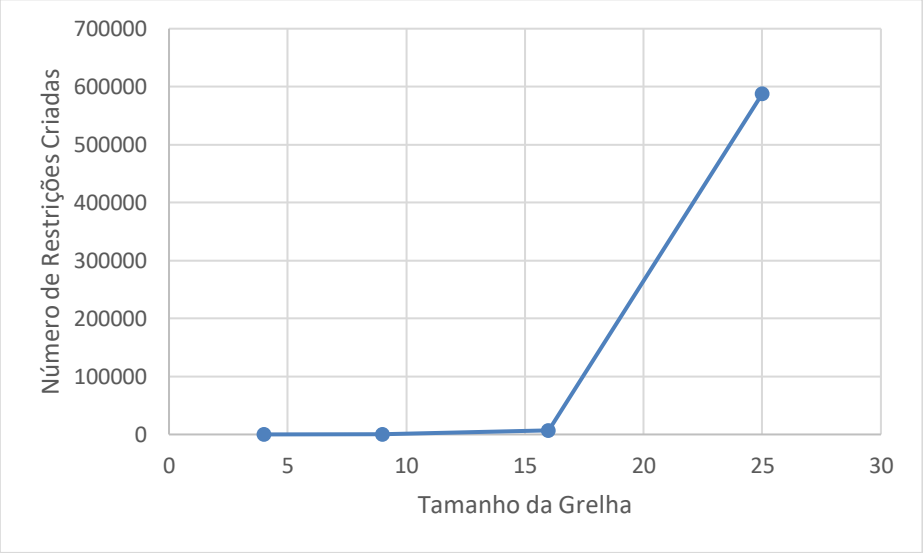


Fig. 6. Variação do número de restrições criadas em função da dimensão da grelha/tabuleiro

8.2 Código Fonte

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

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

printBoard(Board):-
    nl,
    nth1(1, Board, Row),
    length(Row, Length),
    write('|'),
    print_separator(0, Length),
    nl,
    printMatrix(Board, 0, Length).

%prints the current game matrix in a board-like way
printMatrix([], _, _).
printMatrix([H|T], X, Length):-
    write('|'),
    X1 is X+1,
    printLine(H),
    nl,
    write('|'),
    print_separator(0, Length),
    nl,
    printMatrix(T, X1, Length).

%prints a matrixs line
printLine([]).
printLine([H|T]):-
    printElement(H),
    printLine(T).

printElement(H):-
    H < 10,
    write('    '),
    write(H),
    write(' |').
printElement(H):-
    H >= 10,
    H < 100,
    write('   '),
    write(H),
```

```
    write(' |').
printElement(H):-
    H >= 100,
    H < 1000,
    write('  '),
    write(H),
    write(' |').
printElement(H):-
    H >= 1000,
    write(' '),
    write(H),
    write(' |').

print_separator(Length, Length).
print_separator(Aux, Length):-
    write('------|'),
    NewAux is Aux+1,
    print_separator(NewAux, Length).


flatten([], []) :- !.
flatten([L|Ls], FlatL) :-
    !,
    flatten(L, NewL),
    flatten(Ls, NewLs),
    append(NewL, NewLs, FlatL).
flatten(L, [L]).


unflatten([], _, _,_ , []).
unflatten([H|T], N, N, CurrentRow, [NewCurrentRow|Rest]):-
    append(CurrentRow, [H], NewCurrentRow),
    unflatten(T, 1, N, [], Rest).
unflatten([H|T], CurrentN, N, CurrentRow, Final):-
    append(CurrentRow, [H], NewCurrentRow),
    NewCurrentN is CurrentN + 1,
    unflatten(T, NewCurrentN, N, NewCurrentRow, Final).

findSolvableMatrix(Length,Value,Matrix):-
    MatrixSize is Length*Length, % NxN board,
    (
        generateRandomMatrix(MatrixSize,Matrix),
        solution(Matrix,Length,Value,_),!
        )
    ;
    (
        findSolvableMatrix(Length,Value,Matrix)
        ).
```

```
generateRandomMatrix(0, []).
generateRandomMatrix(C, Y):-
  C > 0,
  C1 is C-1,
  random(1, 10, U),
  Y = [U|T],
  generateRandomMatrix(C1, T).

constrainSum([], _). %Solution must have sum equal to Value
constrainSum([H|T], Value):-
    sum(H, #=, Value),
    constrainSum(T, Value).


restriction(Var, Aux):- %Solution must contain digit Aux
    Var mod 10 #= Aux.
restriction(Var, Aux):-
    Var #> 0,
    Rest #= Var // 10,
    restriction(Rest, Aux).

addRestrictions([],_,_).
addRestrictions([H|T], Vars, Index):-
    nth1(Index, Vars, Aux), %Value of the input
    restriction(H, Aux),
    NewIndex is Index + 1,
    addRestrictions(T, Vars, NewIndex).


solution(Input, No, Value, Out-
put):- %Board NxM ????? Only works on NxN boards right?
    unflatten(Input, 1, No, [], Matrix),
    nth1(1, Matrix, Col), % Row = [x,y,z]
    length(Col, N),
    Total is N*N, % NxN board
    length(Output, Total), % Output now has length Total
    domain(Output, 1, Value), % Values from Out-
put in 1..100
    unflatten(Output, 1, N, [], Rows), % Reverse opera-
tion of flatten
    trans-
pose(Rows, Cols), % Gives us Cols, since we need both Rows
and Cols to be = Value
    constrainSum(Rows, Value),
    constrainSum(Cols, Value),
    addRestrictions(Output, Input, 1),
    labeling([], Output).
```

```
cnote(Input, No, Value):-
    unflatten(Input, 1, No, [], Matrix),
    nl, write('Matrix: '), nl,
    printBoard(Matrix),
    nth1(1, Matrix, Row),
    length(Row, Length),
    reset_timer, %start timer before solution
    solution(Input, No, Value, Output),
    unflatten(Output, 1, Length, [], Result),
    write('Result: '), nl,
    printBoard(Result),
    print_time, nl,
    fd_statistics.


cnoteGenerate(Length,Value,PossibleMatrix):-
    findSolvableMatrix(Length,Value,PossibleMatrix).


cnoteRandom(Length,Output):-
    MatrixSize is Length*Length,
    generateRandomMatrix(MatrixSize,Output).
```