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.

2 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.

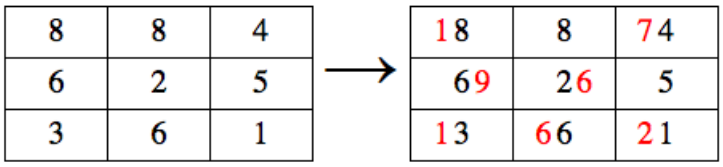

Fig. 1. Exemplo de problema e a sua solução

3 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.

3.1 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**.

- 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**).

3.2 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
  ```

4 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, Origin
alM) :- % 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 módulo 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(' |').
```

5 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.

5.1 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:

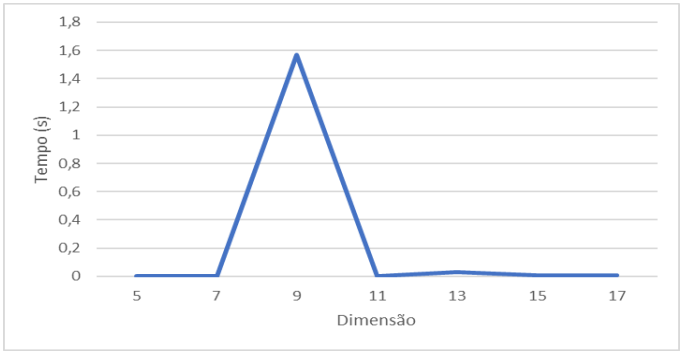

Fig. 2. Tempo de labeling em relação à dimensão

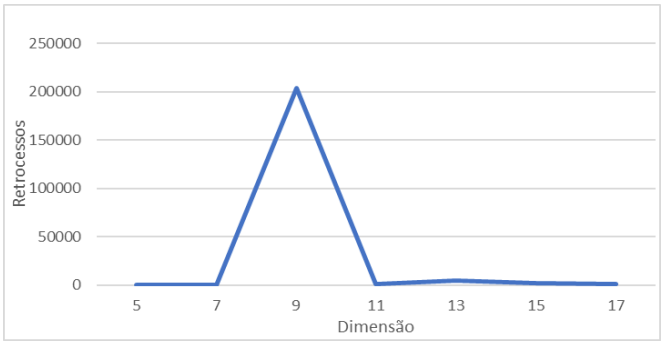

Fig. 3. Número de retrocessos em função da dimensão

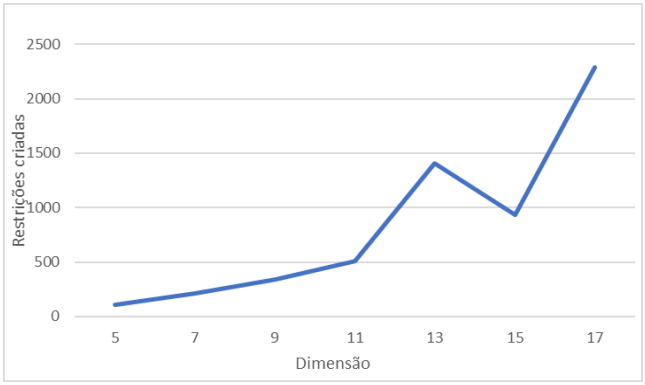

Fig. 4. Número de restrições criadas em função da dimensão

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.

5.2 Estratégias de Pesquisa

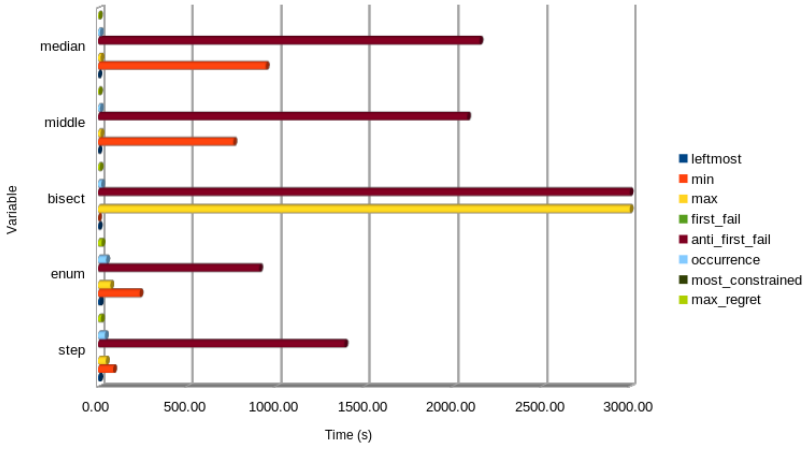

Fig. 5. Gráfico do tempo de labelling nas várias combinações

	leftmost	min	max	first_fail	anti_first_fail	occurrence	most_constrained	max_regret
step	7.34	85.14	43.60	0.00	1389.97	38.24	0.00	16.34
enum	10.42	233.86	69.28	0.00	908.99	45.49	0.00	19.80
bisect	3.89	0.37	3000.00	0.00	3000.00	18.59	0.00	9.38
middle	2.06	763.30	13.96	0.00	2082.89	10.98	0.00	5.14
median	2.11	945.63	14.05	0.00	2152.34	10.88	0.00	4.94

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.

6 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.

7 Anexos

7.1 Código fonte

cnote.pl

```
:- use_module(library(lists)).
:- use_module(library(clpfd)).
:- use_module(library(random)).
:- consult('utils.pl').
:- consult('generate.pl').
:- consult('menu.pl').

% Number unmodified
presentNum(_, _, Number) :-
    Number < 10,
    write('  '), write(Number), write('  |').

% Added Number on the left
presentNum(Original, Original, Number) :-
    write(' '), write(Number), write('  |').

% Added Number on the right
presentNum(_, _, Number) :-
    write('  '), write(Number), write(' |').

% Decrements the line counter
decrementN(N, 0, NewN, OriginalM, OriginalM) :-
    NewN is N - 1, nl.

decrementN(NewN, M, NewN, M, _).

% Present Solution
presentResult([], [], 0, _, _).
presentResult([Original | RestOriginal], [First | Rest], N, M, Origin
alM) :- % 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).

% Get line length and line from input
gridLine('TRUE', LineGrid, M) :-
    write('Enter your Line: '),
    getLine(LineGrid, M).

% Set the length of a line
gridLine('FALSE', LineGrid, M) :-
    length(LineGrid, M).

% Generate an empty grid or grid from input
generateGrid(0-_, Grid, Grid, _) :- !.
generateGrid(N-M, Aux, Grid, ReadInput) :-
    gridLine(ReadInput, LineGrid, M),
    append(Aux, [LineGrid], NewAux),
    NewN is N - 1,
    generateGrid(NewN-M, NewAux, Grid, ReadInput).

% CNote core predicate
cNote(InputGrid, DynamicGrid, DigitGrid, ResultGrid, Flag, Timeout, O
ption) :-
    applyConstraints(InputGrid, DigitGrid, DynamicGrid),
    flattenGrid(DynamicGrid, [], ResultGrid),
    reset_timer,
    labelCNote(ResultGrid, Timeout, Flag, Option).

% CNote core predicate when no solutions where found for the given pu
zzle
cNote(_, _, _, _, nosolutions, _, _).

% Labeling (solver)
labelCNote(ResultGrid, _, _, solvePuzzle) :-
    labeling([ff, ffc], ResultGrid).

% Labeling (generator)
labelCNote(ResultGrid, Timeout, Flag, generatePuzzle) :-
```

```
    labeling([time_out(Timeout, Flag), value(selRandomValue), variabl
e(selRandomVariable)], ResultGrid).

% Apply constraints related to a single cell
applyConstraintsLine([], [], []).
applyConstraintsLine([H|T], [S|T2], [R|T1]) :-
 % Input | Digits | Result
    H in 1..9,
    S in 0..9,
    R in 1..99,
    R #= H*10 + S #\/ R #= S*10 + H,
    applyConstraintsLine(T, T2, T1).

% Apply line constraints
applySumConstraintsLines([], [], []).
applySumConstraintsLines([Prob|ProbRest], [Sol|SolRest], [Line|Rest])
 :-
    sum(Line, #=, 100),
    applyConstraintsLine(Prob, Sol, Line),
    applySumConstraintsLines(ProbRest, SolRest, Rest).

% Apply constrains per column (sum of column has to be 100)
applySumConstraintsColumns(_, 0) :- !.
applySumConstraintsColumns(Grid, Ncol) :-
    applySumConstraintsColumnsAux(Grid, Ncol, [], Column),
    sum(Column, #=, 100),
    NewNcol is Ncol - 1,
    applySumConstraintsColumns(Grid, NewNcol).

% Apply column constrains
applySumConstraintsColumnsAux([], _, Result, Result).
applySumConstraintsColumnsAux([Line | Rest], Ncol, Aux, Result) :-
    element(Ncol, Line, Element),
    append(Aux, [Element], NewAux),
    applySumConstraintsColumnsAux(Rest, Ncol, NewAux, Result).

% Apply all constraints
applyConstraints(Prob, Sol, Res) :- % Input | Digits | Result
```

```
    applySumConstraintsLines(Prob, Sol, Res),
    nth1(1, Prob, Line),
    length(Line, Ncols),
    applySumConstraintsColumns(Res, Ncols).

% Flattens a 2 dimension grid into a 1 dimension grid
flattenGrid([], Result, Result).
flattenGrid([Line|Rest], Aux, Result) :-
    append(Aux, Line, NewAux),
    flattenGrid(Rest, NewAux, Result).

% Solve puzzle given by the user
solveCNote :-
    readPuzzleInput(N, M),
    generateGrid(N-M, [], InputGrid, 'TRUE'), % Read Input Puzzle
    generateGrid(N-
M, [], DynamicGrid, 'FALSE'), % Generate Dynamic List
    generateGrid(N-M, [], DigitGrid, 'FALSE'), % Generate Digit List
    !,
    cNote(InputGrid, DynamicGrid, DigitGrid, ResultGrid, Flag, 5000,
solvePuzzle),
    finalCNote(Flag, InputGrid, ResultGrid, N, M).

% Generate a random puzzle
generateCNote :-
    readPuzzleInput(N, M),
    generateGrid(N-M, [], InputGrid, 'FALSE'), % Read Input Puzzle
    generateGrid(N-
M, [], DynamicGrid, 'FALSE'), % Generate Dynamic List
    generateGrid(N-M, [], DigitGrid, 'FALSE'), % Generate Digit List
    !,
    cNote(InputGrid, DynamicGrid, DigitGrid, ResultGrid, Flag, 5000,
generatePuzzle),
    finalCNote(Flag, InputGrid, ResultGrid, N, M).

% Solve one of the hard coded puzzles
hardCNote :-
    write('Insert the puzzle number: '),
```

```
    getInt(PuzzleN),
    puzzle(PuzzleN, N, M, InputGrid), % Selected Puzzle
    generateGrid(N-
M, [], DynamicGrid, 'FALSE'), % Generate Dynamic List
    generateGrid(N-M, [], DigitGrid, 'FALSE'), % Generate Digit List
    !,
    write(InputGrid),
    cNote(InputGrid, DynamicGrid, DigitGrid, ResultGrid, Flag, 5000,
solvePuzzle),
    finalCNote(Flag, InputGrid, ResultGrid, N, M).

% Prints statistics and the result grid
finalCNote(success, InputGrid, ResultGrid, N, M) :-
    print_time,
    fd_statistics,
    flattenGrid(InputGrid, [], Input),
    presentResult(Input, ResultGrid, N, M, M).

% Displayed when no solutions were found within the given timeout
finalCNote(time_out, _, _, _, _) :-
    nl, write('No solutions found within 5s!'), nl.

% Displayed when no solutions were found
finalCNote(nosolutions, _, _, _, _) :-
    nl, write('No solutions found!'), nl.

% Puzzle start
cnote :-
    mainMenu.
```

utils.pl

```
% Reads an integer with error detection
getInt(Int) :-
    repeat,
    getIntAux(Int).
```

```
getIntAux(Int) :-
    catch(read(Int), _, true),
    read_line(_),
    integer(Int),
    nl.

getIntAux(_) :-
    write('Invalid input! Please try again.'), nl,
    fail.

getNewInt(Int) :-
    (
        catch(read(Int), _, true),
        read_line(_),
        integer(Int),
        nl
    ).

% Reads the dimension grid
getDimension(N) :-
    repeat,
    getDimensionAux(N), !.

getDimensionAux(N) :-
    getInt(N),
    N > 0.

getDimensionAux(_) :-
    write('Invalid dimension. Please try again.'), nl,
    fail.
% Reads a grid line
getLine(Line, M) :-
    repeat,
    getLineAux(Line, M).

% Checks if the grid's line is indeed a list with M length and with v
alues in [1, 9]
```

```
getLineAux(Line, M) :-
    catch(read(Line), _, fail),
    read_line(_),
    Line = [_|_],
    length(Line, M),
    checkValidRow(Line).

getLineAux(_, _) :-
    write('Invalid Line. Please try again.'), nl,
    fail.

% Check if a given row is valid
checkValidRow([]).
checkValidRow([H | T]) :-
    H >= 1 , H =< 9,
    checkValidRow(T).

% Reads a single character
getChar(Char) :-
    get_char(Char),
    read_line(Line),!,
    (Line == ""; Line == "."),
    nl.

% Reset labeling timer
reset_timer :- statistics(walltime,_).

% Print the labeling time on the screen
print_time :-
    statistics(walltime,[_,T]),
    TS is ((T//10)*10)/1000,
    nl, write('Time: '), write(TS), write('s'), nl, nl.

% Get puzzle grid dimension from user input
readPuzzleInput(N, N) :-
    write('Insert Number of Lines/Columns: '),
    getDimension(N).
```

```
% Selects value randomly
selRandomValue(Var, _, BB0, BB1):-
    fd_set(Var, Set), fdset_to_list(Set, List),
    random_member(Value, List),
    ( first_bound(BB0, BB1), Var #= Value ;
    later_bound(BB0, BB1), Var #\= Value ).

% Selects a random variable
selRandomVariable(ListOfVars, Var, Rest):-
    random_select(Var, ListOfVars, Rest).
```

menu.pl

```
% Main menu predicate
mainMenu :-
    repeat,
    (
        checkInput(Input),
        \+manageInput(Input)
    ).

% Checks if menu input is valid
checkInput(Input) :-
    printMainMenu,
    repeat,
    (checkInputAux(Input)), !.

% Checks if inserted option is within the given range
checkInputAux(Input) :-
    askMenuOption,
    getNewInt(Input),
    Input >= 0,
    Input =< 4, !.

% When an error occurs regarding a user input
checkInputAux(_) :-
    nl, write('ERROR: Invalid Option!'), nl, nl,
    fail.
```

```
% Prompt for menu option
askMenuOption :-
    write('> Insert your option ').

manageInput(0, _) :- % Exit
    write('\nExiting...\n\n'), fail.

manageInput(1) :- % Insert puzzle and solve
    solveCNote.

manageInput(2) :- % Generate puzzle dynamically and solve
    generateCNote.

manageInput(3) :- % Choose puzzle and solve
    hardCNote.

manageInput(4) :- % About
    nl,
    write('._____________________________________________________
______.'), nl,
    write('|
      |'), nl,
    write('|                                ABOUT
      |'), nl,
    write('|=========================================================
======|'), nl,
    write('|      Given an initial grid filled with digits from 1 to
9,    |'), nl,
    write('|  the grid must be completed by adding one more digit in
every |'), nl,
    write('|  cell to the right or left of the initial digit, so that
 the  |'), nl,
    write('|      sum of all elements in every row and column is 100.
      |'), nl,
    write('|_____________________________________________________
______|'), nl,
    getChar(_).
```

```
% Main Menu
printMainMenu :-
    nl, nl,
    write(' _____________________________________________________
_______ '), nl,
    write('|
       |'), nl,
    write('|                    ___          __      _
       |'), nl,
    write('|                   / __\\      /\\ \\ \\___ | |_ ___
          |'), nl,
    write('|                  / /  _____ /  \\/ / _ \\| __/ _ \\
         |'), nl,
    write('|                 / /__|_____/ /\\  / (_) | ||  __/
        |'), nl,
    write('|                 \\____/     \\_\\ \\/ \\___/ \\__\\___|
             |'), nl,
    write('|
       |'), nl,
    write('|
       |'), nl,
    write('|                          Davide Castro
       |'), nl,
    write('|                                &
       |'), nl,
    write('|                             Rui Pinto
       |'), nl,
    write('|=====================================================
=======|'), nl,
    write('|
       |'), nl,
    write('|                       1. Insert     Puzzle
       |'), nl,
    write('|                       2. Generate   Puzzle
       |'), nl,
    write('|                       3. Choose     Puzzle
       |'), nl,
```

```
    write('|
       |'), nl,
    write('|                              4. About
       |'), nl,
    write('|                              0. Exit
       |'), nl,
    write('|_____________________________________________________________
_______|'), nl.



% (Ogre)
%
%    ___     __      _
%   / __\ /\ \ \___ | |_ ___
%  / /   /  \/ / _ \| __/ _ \
% / /___/ /\  / (_) | ||  __/
% \____/\_\ \/ \___/ \__\___|
```

generate.pl

```
%  Hard coded Puzzles
puzzle(1, 3, 3,
    [
            [8, 7, 8],
            [3, 1, 2],
            [2, 7, 2]
    ]).

puzzle(2, 3, 3,
    [
        [8, 7, 8],
        [3, 1, 2],
        [2, 7, 2]
    ]).


puzzle(3, 3, 3,
```

```
    [
        [1, 1, 6],
        [2, 7, 9],
        [3, 2, 4]
    ]).


puzzle(4, 3, 3, [
        [7, 5, 1],
        [2, 7, 2],
        [9, 4, 7]
    ]).

puzzle(5, 3, 3, [
        [4, 6, 5],
        [8, 2, 5],
        [7, 1, 8]
    ]).

puzzle(6, 3, 3, [
        [4, 9, 7],
        [6, 2, 4],
        [5, 7, 4]
    ]).

puzzle(7, 3, 3,
    [
        [4, 3, 6],
        [3, 3, 9],
        [4, 3, 7]
    ]).

puzzle(8, 3, 3,
    [
        [1, 4, 7],
        [9, 7, 3],
        [9, 6, 2]
    ]).
```

```
puzzle(9, 3, 3, [
        [7, 3, 9],
        [8, 2, 5],
        [7, 1, 8]
    ]).

puzzle(10, 3, 3,
    [
        [1, 2, 1],
        [2, 7, 8],
        [6, 2, 5]
    ]).

puzzle(11, 3, 3, [
        [9, 1, 8],
        [8, 7, 6],
        [7, 2, 4]
    ]).

puzzle(12, 3, 3,
    [
        [2, 8, 2],
        [1, 4, 7],
        [9, 5, 6]
    ]).

puzzle(13, 4, 4,
    [
        [4,6,7,7],
        [8,2,1,9],
        [3,3,5,9],
        [9,9,7,5]
    ]).

puzzle(14, 4, 4,
    [
        [4,1,3,2],
```

```
        [3,6,2,9],
        [2,2,3,7],
        [3,3,2,2]
    ]).

puzzle(15, 4, 4,
    [
        [1,6,4,9],
        [3,4,7,9],
        [6,8,8,8],
        [3,2,1,4]
    ]).

puzzle(16, 5, 5,
    [
        [7,7,7,2,7],
        [3,2,4,6,5],
        [8,8,7,5,7],
        [1,2,8,7,3],
        [2,1,4,5,8]
    ]).
```