Resolução de um Problema de Decisão usando Programação em Lógica com Restrições – Puzzle Trid

Igor Silveira [up201505172] e João Almeida [up201504874]

FEUP-PLOG, Turma 3MIEIC04, Grupo Trid_4.

Resumo. Este trabalho pretende demonstrar a eficiência da resolução de um puzzle através dos conceitos adquiridos de programação em lógica com recurso a restrições. O problema de decisão proposto é o Trid, uma variação do Sudoku que consiste no preenchimento de uma tabela, de forma triangular, de modo a que não haja repetição de números numa determinada linha. Este artigo vem demonstrar a solução desenvolvida para a resolução do problema, de forma dinâmica, com utilização do *software* SICStus Prolog.

1 Introdução

Este trabalho foi desenvolvido no âmbito da Unidade Curricular Programação em Lógica, do 1º semestre do 3º ano do Mestrado Integrado em Engenharia Informática e Computação com o principal objetivo de aprofundar os conceitos abordados nas aulas sobre programação em lógica com uso de restrições. Na prática, o grupo utilizou a biblioteca **clpdf** presente no SICStus Prolog como auxiliar à resolução do problema escolhido pelo grupo.

O problema escolhido a partir da oferta proposta foi o puzzle **Trid**, uma variação do Sudoku que partilha o mesmo objetivo comum: não repetir o mesmo número ao longo de uma linha (quer diagonal, quer horizontal). No Trid, a tabela apresenta uma fórmula triangular, composta por vários triângulos interiores e cada vértice desse triângulo será um nó onde é colocado o número.

Este artigo pretende resumir a abordagem que o grupo teve perante a resolução deste problema. Desta forma, parte inicial encontra-se a descrição do problema, explicando o problema em detalhe, seguido da abordagem realizada pelo grupo: as variáveis de decisão e restrições usadas, a função de avaliação e estratégia de pesquisa. No final, encontra-se a visualização da solução obtida, bem como resultados e conclusões acerca da abordagem tomada. Por último, encontra-se a bibliografia utilizada para a realização do projeto bem como o código fonte do programa em anexo.

2 Descrição do Problema

O problema escolhido pelo grupo para a aplicação dos conceitos relacionados com a resolução de problemas de decisão é um puzzle numérico, variante do Sudoku, chamado **Trid**. O objetivo deste puzzle é, dado um certo intervalo (que determina o tamanho do tabuleiro) preencher os nós (vértices de cada triângulo interior) de modo a que nenhum número seja repetido na linha onde se situa, quer diagonal, quer horizontalmente. Os números dentro dos triângulos correspondem à soma dos dígitos dos vértices do mesmo. Na figura 1, encontra-se um exemplo de um **Trid** com um intervalo **[1,5]**, antes e após a resolução.

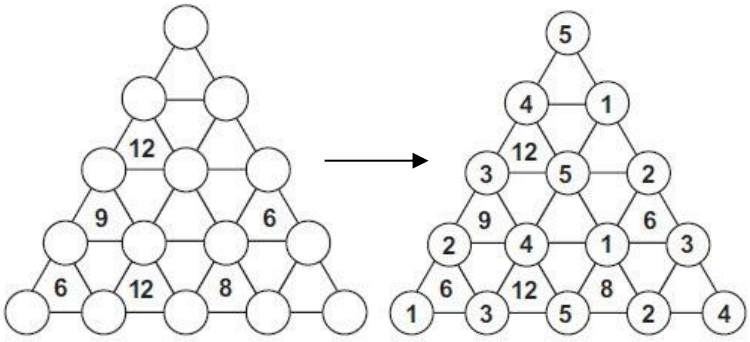

Figura 1 – Trid de base 5, antes da resolução (à esquerda) e resolvido (à direita)

3 Abordagem

Em primeiro lugar foi necessário avaliar o puzzle e tentar perceber como o modelar como um problema de restrições.

Após chegarmos a conclusões sobre esse assunto de seguida focamo-nos numa representação prática e intuitiva para o utilizador de forma a que a solução apresentada fosse facilmente entendida.

3.1 Variáveis de Decisão

A solução pretendida para o puzzle que nos foi atribuído é uma representação triangular de base de tamanho N em que todos os N(N + 1) / 2 nós apresentam valores entre 1 e N sem que estes se repitam nas suas diagonais ou linhas horizontais. Para todo o caso o resultado é uma lista dos N(N + 1) / 2 números que respeitam as regras inerentes ao puzzle e devidamente ordenados para serem dispostos pela Trid. Deste modo, a única variável de decisão que o nosso puzzle necessita, e que é usada no predicado de *labeling*, é uma variável **Valores**.

3.2 Restrições impostas à variável Valores

Para resolver o Trid, foi desenvolvido o predicado **resolvePuzzle**, que aceita um número inteiro inicial maior ou igual a 3 - o tamanho da base do Trid que pode ser escolhido pelo utilizador ou gerado aleatoriamente. As restrições são impostas a todos os nós individualmente através dos predicados **constrainLinhas** e **constrainDiagonais** que irão, respetivamente, criar as restrições linha a linha para cada nó e a toda a diagonal para cada nó. Cada um destes dois últimos predicados recebe a variável de decisão **Valores** que contém todas as N(N + 1) / 2 variáveis a ser restringidas, bem como o tamanho do Trid a resolver e dois contadores inicializados a 1, **Contador = 1** e **K = 1**.

Para ambos os predicados, os **Valores** são percorridos linha a linha, ou seja, é passado 1 nós, de seguida 2 nós, depois 3 e assim sucessivamente, isto é conseguido através das variáveis **Contador** e **K**, sendo que **K** representa o número de nós da linha atual e Contador controla a quantidade de vezes que o predicado intermédio para cada um dos predicados de restrição é executado para essa linha.

Após análise do puzzle e das restrições necessárias, estes predicados diferem entre si no facto em que o predicado de **constrainLinhas** não executa nos últimos nós de cada linha - as restrições horizontais para o ultimo nó já estão cobertas pelos seus vizinhos horizontais - e o predicado de **constrainDiagonais** não executa nos nós da ultima linha - pois estes não têm nós inferiores a si -, isto foi pensado de modo a **evitar redundância** e **aumentar eficiência**.

3.3 Estratégia de Pesquisa

A estratégia de etiquetagem utilizada foi a *default* de **leftmost** e **up**, pois decidimos que para este puzzle estas eram a mais indicadas para obter uma primeira solução possível para apresentar ao utilizador, visto que é também a mais eficiente.

4 Visualização da solução

Depois do ficheiro trid.pl ser consultado no SICStus Prolog, basta escrever **trid.** para se iniciar a execução do programa e aparecer o menu principal.

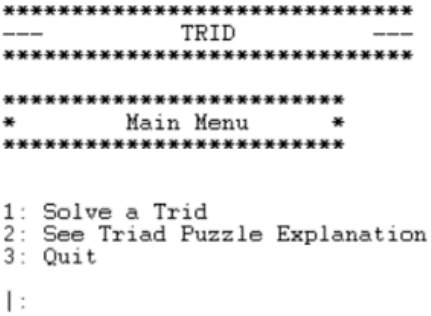

Figura 2 - Menu principal da aplicação

A partir do menu principal pode-se escolher três opções entre a resolução de um Trid, a explicação do puzzle ou sair da aplicação. Escolhendo a primeira opção, é possível escolher a resolução de um Trid com tamanho aleatório (até o limite de 11, que durante os nossos testes revelou durar um tempo aceitável para resultados no "imediato") ou então indicar o tamanho do Trid que queremos resolver. O tamanho estabelecerá o intervalo de números que o Trid usará, sempre entre [1,tamanho].

A representação em modo texto de dois Trids de intervalo [1,9] e [1,4] resolvidos e os respetivos tempos de resolução pode ser visto na figura 3.

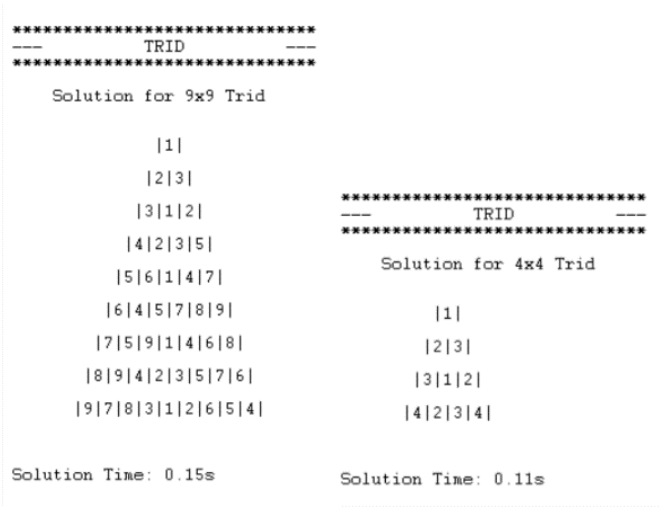

Figura 3 - Resolução do puzzle de tamanhos 9x9 e 4x4

Tendo em conta a nossa abordagem ao problema, foi omitido a soma dos vértices dos triângulos interiores. No final de cada resolução é possível voltar ao menu inicial e voltar a fazer a resolução de Trids com outros tamanhos.

5 Resultados

Após resolução do puzzle o utilizador pode verificar *on-screen* o tempo levado pelo cálculo da solução. Com o aumento do tamanho da base do Trid podemos concluir que tanto o tempo de resolução como a quantidade de restrições criadas aumentam de forma acentuada como se pode verificar na tabela apresentada abaixo.

Tamanho do Trid	Tempo de Execução (s)	Nº Restrições
3	0.03	12
4	0.06	26
5	0.07	60
7	0.09	168
11	5.31	660
12	790.17	…

Tabela 1 - Tabela dos resultados obtidos para Trids de vários tamanhos, confirmando que o tempo de execução aumenta à medida que a dificuldade do puzzle aumenta

Estes testes foram realizados num equipamento 64bits com um Intel Core i5-7200U @ 2.5 – 2.7 GHz e 8GB de RAM.

6 Conclusões e Trabalho Futuro

Após o término do trabalho proposto foi-nos possível concluir que a linguagem de Programação Lógica com a utilização de restrições é uma ferramenta muito importante e facilita bastante a resolução de problemas com algum grau de dificuldade.

Podemos realçar a vantagem desta linguagem de poupar à mente humana a resolução de problemas de decisão/otimização que envolveriam uma quantidade exaustiva de tempo e esforço mental. A principal desvantagem que podemos apontar é a listagem das restrições, um processo algo moroso e em problemas de maior dificuldade pode dar aso a engano (ao não cobrir todas as restrições necessárias).

Tendo que o objetivo proposto foi alcançado, passando pela assimilação de conceitos relacionados com a matéria como declaração de variáveis, de domínios e de restrições, bem como a pesquisa de soluções e o respetivo *labeling* e utilizando como auxiliar a biblioteca **clpfd** do SICStus Prolog, os elementos do grupo estão orgulhosos pelo resultado obtido e ambientados a um novo paradigma de programação.

7 Bibliografia

1. Página da Unidade Curricular no Moodle, https://moodle.up.pt/course/view.php?id=638, último acesso a 23/12/2017.
2. The World of Logical Puzzles, http://rohanrao.blogspot.pt/2009/05/rules-of-trid.html, ultimo acesso a 23/12/2017.
3. Sachsentext, http://www.sachsentext.de/en/taxonomy/term/403, último acesso a 23/12/2017.
4. Prolog: Create sublist, given two indices – StackOverflow, https://stackoverflow.com/questions/16427076/prolog-create-sublist-given-two-indices, ultimo acesso a 23/12/2017.
5. Constraint Logic Programming over Finite Domains—library(clpfd), https://sicstus.sics.se/sicstus/docs/4.1.0/html/sicstus/lib_002dclpfd.html, ultimo acesso a 23/12/2017.

8 Anexo – Código Fonte do Projeto

Trid.pl

```
:- include('Utilitarios.pl').
:- include('Menus.pl').
:- include('Interface.pl').
:- include('Logica.pl').

trid :-
  limpaEcra,
     menuPrincipal.
```

Utilitarios.pl

```
:- use_module(library(lists)).
%- Utilitarios ao Jogo -%

novaLinha(Vezes) :-
    novaLinha(0, Vezes).

novaLinha(Linha, Limite) :-
    Linha < Limite,
    LinhaInc is Linha + 1,
    nl,
    novaLinha(LinhaInc, Limite).

novaLinha(_,_).

limpaEcra :- novaLinha(50), !.

esperaTecla :-
  write('\nEnter anything and "." + ENTER to continue > '),
  read(_Choice).

extraiNumeroLista([H | _T], Numero) :-
  Numero is H.

getNumber(NumberIn, NumberFinal) :-
   char_code(Number1, NumberIn),
   atom_chars(Number1, Number2),
   number_chars(Number, Number2),
   NumberFinal is Number.


ite(If, Then, _) :-
  If, !, Then.

ite(_, _, Else) :-
  Else.

tamanhoLista(Lista, Conta):-
     X = _,
        auxiliarTamanho(Lista, X),
     Conta is X.
```

```
auxiliarTamanho([],X) :-
   X = 0.

auxiliarTamanho([ _ | Cauda ], Conta):-
        auxiliarTamanho(Cauda,Anterior),
        Conta = Anterior + 1.

tamanhoTabuleiro([H | T], XLimite, YLimite) :-
     tamanhoLista(H, XLimite),
     tamanhoLista(T, Tamanho),
     YLimite is Tamanho + 1.

subLista(L, M, N, S) :-
    findall(E, (nth1(I, L, E), I >= M, I =< N), S).

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.
```

Interface.pl

```
%++++++++++++ Tabuleiro ++++++++++++%
imprimeLinha(_,_, S, S) :-
   write('\n').

imprimeLinha([H | T], Y, 0, S) :-
   write(H),
   write('|'),
   imprimeLinha(T, Y, 1, S).

imprimeLinha([H | T], Y, X, S) :-
   X < S,
   write(H),
   write('|'),
   A is X + 1,
   imprimeLinha(T, Y, A, S).

imprimeLinha(_, A, A, _) :-
   write('\n').



imprimeSeparador(X, S) :-
   X < S,
   write(--),
   A is X + 1,
   imprimeSeparador(A, S).

imprimeSeparador(A, A):-
   write('\n').

imprimeTabuleiro([H | T], X, XLimite, YLimite, Espacos):-
```

```
   X < YLimite,
   espaco(-5, Espacos),
   write(' |'),
   imprimeLinha(H, X, 0, XLimite),
   espaco(-5, Espacos),
   %imprimeSeparador(-1, XLimite), %--desde o indice -1, para cobrir os numeros
das linhas--%
   X1 is X + 1,
   imprimeTabuleiro(T, X1, XLimite, YLimite, Espacos).

imprimeTabuleiro(_, _, _, _, _).

imprimeTabuleiro(Tabuleiro, Espacos) :-
   tamanhoTabuleiro(Tabuleiro, XLimite, YLimite),
   nl,
   %imprimeSeparador(-1, XLimite),
   imprimeTabuleiro(Tabuleiro, 0, XLimite, YLimite, Espacos).

imprimeLista(Valores, Inicio, K, Tamanho) :-
   K =< Tamanho,
   Fim is Inicio + K - 1,
   subLista(Valores, Inicio, Fim, Out),
   %write('Linha '), write(K), write(': '),
   Espacos is Tamanho - K,
   ParaImprimir = [Out],
   imprimeTabuleiro(ParaImprimir, Espacos),
   %write(Out), nl,
   K1 is K + 1,
   Inicio1 is Inicio + K,
   imprimeLista(Valores, Inicio1, K1, Tamanho).

imprimeLista(_, _, _, _).

espaco(Limite, Limite).

espaco(K, Limite) :-
   write(' '),
   K1 is K + 1,
   espaco(K1, Limite).
```

Logica.pl

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

calculaSolucao(Valores, _Tamanho) :-
  _Celulas is floor((exp(_Tamanho, 2) + _Tamanho) / 2),
  length(Valores, _Celulas),
  domain(Valores, 1, _Tamanho),
  constrainLinhas(Valores, 1, 1, _Celulas, _Tamanho),
  constrainDiagonais(Valores, 1, 1, _Celulas, _Tamanho),
  labeling([], Valores).

%corre linha a linha - ex nÃ³s 1, 2, 4, 7
constrainLinhas(Valores, K, Index, Celulas, Tamanho) :-
  Index =< Celulas - (Tamanho - 1),
```

```
  Index1 is Index + K,
  K1 is K + 1,
  restricaoLinhaCompleta(Valores, 1, K, Index),
  constrainLinhas(Valores, K1, Index1, Celulas, Tamanho).

constrainLinhas(_, _, _, _, _).

%percorre todos os K nÃ³s da linha
restricaoLinhaCompleta(_, _, 0, _).

restricaoLinhaCompleta(Valores, Contador, K, Index) :-
  nth1(Index, Valores, V1), %no a ser tratado
  Next is Index + 1,
  auxLinhaCompleta(V1, Valores, 1, K, Index), %constringe a linha
  Contador1 is Contador + 1,
  K1 is K - 1,
  restricaoLinhaCompleta(Valores, Contador1, K1, Next).

auxLinhaCompleta(_, _, Final, Final, _).
auxLinhaCompleta(V1, Valores, Contador, K, Index) :-
  Contador1 is Contador + 1,
  NextRight is Index + Contador,
  nth1(NextRight, Valores, V2),
  V1 #\= V2,
  auxLinhaCompleta(V1, Valores, Contador1, K, Index).

constrainDiagonais(_, Tamanho, _, _, Tamanho).
constrainDiagonais(_, _, Celulas, Celulas, _).
constrainDiagonais(Valores, K, Index, Celulas, Tamanho) :-
  K < Tamanho,
  Index1 is Index + K,
  K1 is K + 1,
  safeLine(Valores, 0, K, Index, Celulas, Tamanho),
  constrainDiagonais(Valores, K1, Index1, Celulas, Tamanho).

%percorre todos os K nÃ³s da linha
safeLine(_, K, K, _, _, _).
safeLine(Valores, Contador, K, Index, Celulas, Tamanho) :-
  Index < Celulas - (Tamanho - 1),
  nth1(Index, Valores, V1), %no a ser tratado
  Next is Index + 1,
  safeDiagonal(V1, Valores, K, Index, Index, Tamanho), %constringe os diago-
nais
  Contador1 is Contador + 1,
  safeLine(Valores, Contador1, K, Next, Celulas, Tamanho).

safeDiagonal(_, _, Final, _, _, Final).
safeDiagonal(V1, Valores, K, IndexLeft, IndexRight, Final) :-
  NextLeft is IndexLeft + K,
  NextRight is IndexRight + K + 1,
  nth1(NextLeft, Valores, V2),
  nth1(NextRight, Valores, V3),
  V1 #\= V2, V1 #\= V3,
  K1 is K + 1,
  safeDiagonal(V1, Valores, K1, NextLeft, NextRight, Final).

resolvePuzzle( Tamanho) :-
  imprimeTitulo,
```

```
  nl,
  write('Solving '), write(_Tamanho) ,write('x'), write(_Tamanho) , write('
Trid'), nl,
  reset_timer,
  calculaSolucao(Valores, _Tamanho),
  mostraResolucao(_Tamanho, Valores),
  print_time,
  esperaTecla, menuPrincipal.

mostraResolucao(_Tamanho, Valores) :-
  imprimeTitulo,
  write('    Solution for '), write(_Tamanho), write('x'),
  write(_Tamanho), write(' Trid'),
  novaLinha(2),
  imprimeLista(Valores, 1, 1, _Tamanho),
  %write(Valores),
  novaLinha(2).
```

Menus.pl

```
:- use_module(library(system)).

imprimeTitulo :-
   limpaEcra,
   novaLinha(3),
   write('******************************\n'),
   write('---          TRID          ---\n'),
   write('******************************\n'),
   novaLinha(1).

imprimeExit :-
   nl,
  write('******************************\n'),
  write('--- Now Exiting, goodbye! ---\n'),
  write('******************************\n'),
  novaLinha(2).

explanation :-
   imprimeTitulo,
  write('******************************\n'),
  write('---      HOW TO PLAY      ---\n'),
  write('******************************\n'),
  novaLinha(2),
    write('The Trid puzzle is a triangular puzzle that is quite similar to Su-
doku\n'),
   write('in any given range, you must place numbers in each inner triangle
vertex\n'),
   write('such that no digit is repetead along any straight line.\n'),
   write('In our approach, you can choose the PC to solve a randomly sized Trid
puzzle,\n'),
   write('or specify its size, after selecting the option one "Solve a Trid
puzzle".\n'),
   write('The output is a Trid puzzle solved.\n'),
  novaLinha(2).

%- Menu Inicial -%

menuPrincipal :-
```

```
  imprimeTitulo,
  write('************************\n'),
  write('*        Main Menu      *\n'),
  write('************************\n'),
  novaLinha(2),
  write('1: Solve a Trid\n'),
  write('2: See Triad Puzzle Explanation\n'),
  write('3: Quit\n\n'),
  repeat,
  read(Choice),
  verificaMenu1(Choice).

verificaMenu1(1) :-
  limpaEcra,
  selecionaTipo.

verificaMenu1(2) :-
  limpaEcra,
  explanation,
  esperaTecla,
  menuPrincipal.

verificaMenu1(3) :-
  imprimeExit,
  abort.

verificaMenu1(_) :-
  nl,
  write('[ERROR] Invalid option, please choose between 1 and 3\n\n'),
  false.

% Trid Menu %

selecionaTipo :-
  imprimeTitulo,
  write('*************************\n'),
  write('*      Select a Type     *\n'),
  write('*************************\n'),
  novaLinha(2),
  write('1: Random Size\n'),
  write('2: Custom Size\n\n'),
  repeat,
  read(Choice),
  verificaMenu2(Choice).

verificaMenu2(1) :-
  limpaEcra,
  random(3, 12, Tamanho), %entre 3 e 11
  resolvePuzzle(Tamanho).

verificaMenu2(2) :-
  limpaEcra,
   selecionaTamanho.



verificaMenu2(_) :-
   nl,
```

```
  write('[ERROR] Invalid option, please choose between 1 and 2\n\n'),
  false.

%- Size Menu -%

selecionaTamanho :-
  imprimeTitulo,
  write('**************************\n'),
  write('*     Select a Number    *\n'),
  write('**************************\n'),
  novaLinha(2),
  write('[NOTICE] The smallest TRID allowed is of 3x3\n'),
  novaLinha(2),
  repeat,
  read(Choice),
  verificaMenu3(Choice).

  verificaMenu3(1) :-
    nl,
    write('\n[ERROR] The smallest TRID allowed is of 3x3\n\n'),
    !,
    false.

  verificaMenu3(2) :-
    nl,
    write('\n[ERROR] The smallest TRID allowed is of 3x3\n\n'),
    !,
    false.

  verificaMenu3(Tamanho) :-
    limpaEcra,
    resolvePuzzle(Tamanho).
```