Resolução de um Problema de Decisão usando Programação em Lógica com restrições - Pinwheel Puzzle

Pedro Martins (201005350) e Pedro Fraga (201303095)
FEUP-PLOG, Turma 3MIEIC1, Grupo Pinwheel_Puzzle_5.

Resumo: Com este trabalho pretende-se demonstrar a eficácia da utilização de programação em lógica com restrições para resolver problemas de decisão. O problema de decisão proposto é baseado no enigma Pinwheel Puzzle, que consiste num tabuleiro redondo composto por quatro níveis e nove partições, sendo o objetivo ordenar cada partição por forma a que o somatório em cada uma delas seja vinte (ou quinze, se excluirmos o nível mais externo). Através da manipulação de predicados disponibilizados pelo *SICStus Prolog,* mostramos neste artigo que foi não só possível resolver o problema em específico, como, de forma bastante eficiente, expandir e resolver o problema para qualquer outro caso mais geral.

1 Introdução

Este projeto, desenvolvido no âmbito de Programação em Lógica, tem como principal objetivo dotar os estudantes da capacidade de elaborar a resolução de um problema de decisão, utilizando para isso conceitos de programação em lógica com restrições (através do uso da biblioteca '*clpfd*' presente no *SICStus-Prolog*). O nosso grupo, após ponderar sobre todos os temas propostos, decidiu escolher o *Pinwheel Puzzle*, um problema de decisão, que se mostrou intuitivo, com regras fáceis de perceber, e com a possibilidade de expansão para um problema mais abrangente, que permitisse uma resolução mais geral.

O *Pinwheel Puzzle*, no seu problema mais específico, é um enigma cujo objetivo é a ordenação das peças que o constitui. O *puzzle* consiste num tabuleiro circular com quatro níveis diferentes, dividido em nove partições, sendo cada partição constituída por quatro números. Isto dá um total de 36 números (entre 1 e 10, excluindo 9), que devem ser arranjados de forma a que o somatório de cada partição totalize 20.

Este artigo pretende documentar a abordagem que o grupo utilizou para atingir o resultado pretendido, explicando o problema em detalhe e a sua resolução, demonstrando os frutos obtidos e a visualização dos mesmos, bem como havendo ainda espaço para uma conclusão e a bibliografia.

2 Descrição do Problema

O desafio proposto para resolução é um problema de decisão chamado *Pinwheel Puzzle*. Este *puzzle,* cujo formato encontra-se na *Fig* 1, possui quatro anéis circulares coincidentes, cada um deles composto por nove números. Em cada anel, os números podem misturar-se entre si (exceto no anel mais externo, cujos números são estáticos e não se podem mover). O tabuleiro possui ainda nove partições, sendo cada uma dessas partições constituída por quatro números (um de cada anel), resultando num total de 36 números diferentes.

O objetivo deste problema é ordenar as peças em cada um dos três anéis mais internos, de forma a que a soma em cada partição seja 20. Se excluirmos o anel mais externo, existe ainda outra versão do *puzzle*, cuja finalidade é fazer com que o somatório de cada partição seja 15.

Fig. 1. Pinwheel Puzzle por resolver, com as peças desordenadas. Duas peças podem ser trocadas dentro do seu próprio anel.

De forma a tornar o problema mais abrangente, e com o propósito de mostrar a capacidade do paradigma de Programação em Lógica com restrições para resolver, com eficiência, problemas deste género, o grupo decidiu, ainda, expandir o *puzzle* para o caso mais geral, abstraindo-se completamente do formato do mesmo e modelando-o como um *PSR*. Isto permitiu focar o objetivo deste projeto na tentativa de dar uma resposta à seguinte pergunta: de que forma podemos ordenar um conjunto desordenado de números num conjunto de linhas de modo a que, quando ordenados, a soma de todas as colunas dê sempre o mesmo número?

3 Abordagem

O primeiro passo na abordagem foi tentar perceber como modelar o *puzzle* como um problema de restrições. O grupo debruçou-se e empenhou-se em entender as variáveis de decisão a utilizar no predicado de *labeling*, a forma mais correta de restringir essas variáveis, medidas para expandir o problema para um caso menos específico, métodos para testar os resultados obtidos e a maneira mais intuitiva de interagir com o utilizador.

3.1 Variáveis de decisão

A solução pretendida para este *puzzle* é o próprio tabuleiro com os valores todos ordenados, de forma a que a soma de cada partição ou coluna dê 20. Mesmo no caso mais geral, independentemente do número de colunas, do número de linhas ou do valor que cada coluna deve ter quando os valores que a constituem são somados, o resultado é sempre uma lista de listas (ou seja, uma matriz), com os valores já devidamente misturados, respeitando todas as regras inerentes ao problema. Neste sentido, a única variável de decisão (ou variável de domínio) que o nosso problema necessita para ser resolvido, e a única utilizada no nosso predicado *labeling*, é uma variável chamada **Results**.

3.2 Restrições impostas à variável Results

Para resolver o problema proposto, desenvolveu-se um predicado chamado *solver*, que aceita um *Board* inicial – o conjunto de listas que compõe cada linha da matriz – e uma variável chamada *Sum*, que corresponde ao valor do somatório das colunas que compõe a matriz. As restrições são impostas aos valores de cada linha, através da restrição **sorting/3**, e aos valores de cada coluna, através do predicado ***sum/3***, disponibilizado pela biblioteca '*clpfd*'. Este predicado aceita uma lista de inteiros (neste caso, a própria lista que representa a coluna), uma relação operacional e o valor a ser usado por essa relação (neste caso, a restrição aritmética '**#=**' e o valor em ***Sum***). De modo a responder àquilo que o *puzzle* original pedia, consideraram-se as seguintes listas iniciais, correspondentes às linhas do *puzzle* (e que são, portanto, passadas como argumento ao predicado *solver*):

```
firstList([3, 4, 5, 1, 10, 8, 6, 5, 3]).
secondList([5, 4, 6, 4, 6, 6, 4, 5, 5]).
thirdList([5, 7, 8, 4, 5, 3, 2, 6, 7]).
fourthList([8, 5, 6, 2, 1, 4, 6, 7, 4]).
```

Este predicado *solver* vai ter o seguinte comportamento:

- Chama o predicado *sortBoard* com a matriz por resolver, que vai organizar, de forma ascendente, e a partir do predicado do *SICStus samsort/2*, as lista de inteiros que constituem essa matriz.
- A matriz composta pelas listas ordenadas é, então, usada pelo predicado *permutate*, que fará uso da **restrição combinatória *sorting/3***, a primeira a ser usada no nosso trabalho. A matriz ordenada, colocada como terceiro argumento na restrição, irá garantir que o domínio de cada valor (em cada linha) de uma possível solução que respeite as regras do problema, só pode ser uma permutação possível dos valores nessa mesma linha. Assim, considerando, por exemplo, a lista ordenada [1, 2, 3, 5, 7], o predicado *sorting/3* irá garantir que o domínio de uma possível lista resultante varia entre 1 e 7, derivando de uma permuta de até 5 valores diferentes. Isto significa que vai aceitar, por exemplo, [3, 7, 1, 5, 2], mas não [1, 3, 5, 7, 9] (porque 9 não existe na lista original).
- De seguida, uma vez definido um conjunto de linhas válido, importa trabalhar nas colunas, resolvendo o problema principal de fazer com que a soma dos valores que as constituem seja o mesmo. É usado então o predicado *transpose*, que, tal como o nome indica, devolve a matriz transposta, de modo a que as linhas se transformem em colunas (e vice-versa).
- É utilizada a já referida **restrição aritmética *sum/3***, através do predicado *sumBoard*, para garantir que o somatório das colunas é igual ao valor passado como argumento ao predicado *solver*.
- Um outro predicado chamado *flattenList* retira qualquer lista de dentro de outras listas, garantindo que todos os valores da matriz são colocados numa única e última lista, para ser usada como argumento do predicado ***labeling/2***.

3.3 Gerador aleatório do problema a resolver.

O nosso projeto, além de dois modos que permitam ao utilizador resolver o problema do *Pinwheel Puzzle* na sua versão original, dispõe ainda de uma outra funcionalidade, que abrange o problema ao nível mais geral. Isto implica que o utilizador não fica limitado às dimensões originais do *puzzle* (tais como o tamanho de 4 linhas por 9 colunas, o total da soma nessas colunas ou o conjunto de valores disponíveis para trabalhar), podendo, livremente, pedir ao programa que gere um problema completamente aleatório, precisando para isso apenas de fornecer o valor que deseja para a soma nas colunas (de entre 10 a 40). O utilizador pode, depois, tentar resolver o problema por si mesmo (no papel) ou pedir ao próprio programa para encontrar todas as soluções possíveis, até ficar satisfeito com alguma.

O puzzle aleatório inclui um número de colunas e linhas também aleatório entre 4 e 10. O grupo decidiu restringir o tamanho para que não haja puzzles

excessivamente grandes. Depois disso, é calculada uma solução para o problema cuja soma entre colunas resulte no número especificado anteriormente. Este cálculo contém também uma restrição de maximização de números diferentes no puzzle, para que não haja somas óbvias e iguais entre colunas, e uma opção de pesquisa "*time-out*" para que calcule outro tamanho aleatório, caso esteja a demorar a encontrar uma solução para o tamanho outrora definido. Depois de ser calculado um *puzzle* resolvido (isto para ter a certeza que o *puzzle* que é apresentado ao utilizador é, efetivamente, possível de ser resolvido), é altura de misturar todas as linhas para que este se torne num *puzzle* válido, mas por resolver.

4 Visualização da solução

O utilizador interage com o programa iniciando-o através do predicado *start*. De seguida, o mesmo pode escolher um de entre três modos disponíveis:

- Gerar um problema no modo 'Soma 20', que cria um problema aleatório do *Pinwheel Puzzle* com o anel externo.
- Gerar um problema no modo 'Soma 15', que cria, nos mesmos moldes, uma versão do *Pinwheel Puzzle*, desta vez excluindo o anel externo.
- Gerar um problema com soma definida, que pede ao utilizador um valor entre 10 e 40 e gera aleatoriamente uma versão do *puzzle* cujo objetivo é ordenar as linhas que o constituem, de forma a que os valores nas colunas resultem nesse tal valor.

```
                + - + - + - + - + - + - + - + - + - + - + - +

                - - >   P I N W H E E L   P U Z Z L E   < - -

                + - + - + - + - + - + - + - + - + - + - + - +

Desenvolvido por Pedro Fraga e Pedro Martins, no ambito de Programacao em Logica.
Escolha por favor o que deseja fazer:

1 - Gerar um problema no modo 'soma 20'.
2 - Gerar um problema no modo 'soma 15'.
3 - Gerar um problema aleatorio com soma definida.
4 - Ligar as estatisticas.
5 - Sair do programa.

Opcao (entre 1 e 5) =
```

Fig. 2. Menu inicial, acedido através do predicado 'start'.

Gerar um problema no modo 'soma 20' ou no modo 'soma 15' vai gerar um problema aleatório da versão comercial do *Pinwheel Puzzle* (com os valores devidamente misturados em cada linha).

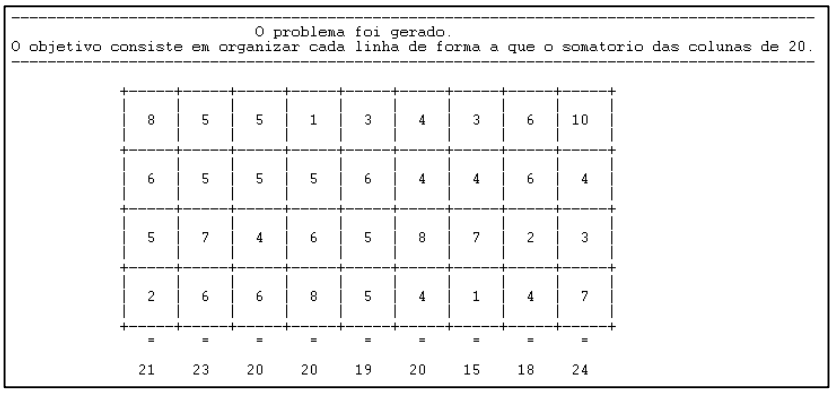

Fig. 3. Problema gerado aleatoriamente para o modo ‘soma 20’.

De forma a tornar o programa mais intuitivo, e devido à dificuldade de implementar e visualizar um tabuleiro circular, o nosso grupo optou por se abstrair desse conceito, imprimindo em modo de texto um clássico tipo de tabuleiro baseado numa matriz (como se mostra na Fig. 3.). Os anéis podem ser entendidos como linhas e as partições do círculo entendidas como colunas.

O utilizador pode escolher encontrar uma solução para o problema, gerar uma nova versão do problema ou voltar ao menu inicial.

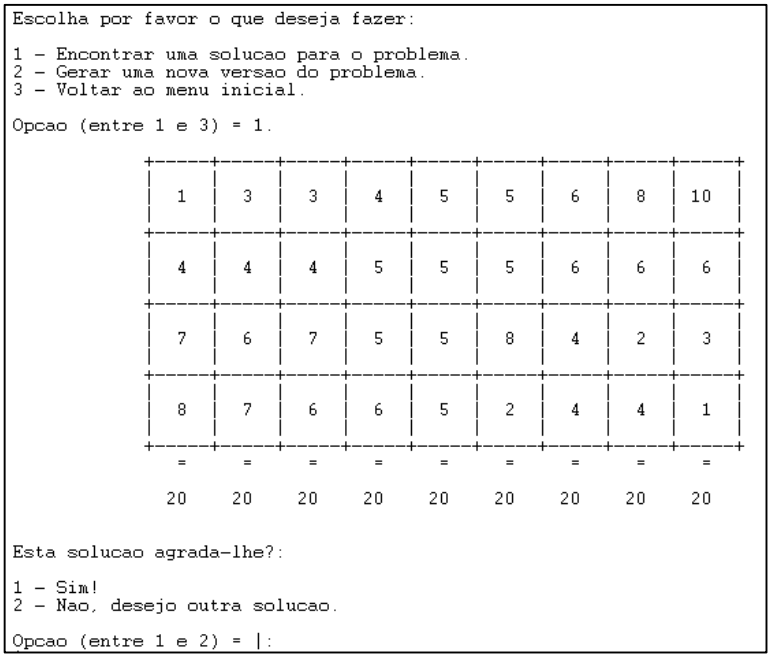

Fig. 4. . Uma solução encontrada para o problema ‘soma 20’. O utilizador pode continuamente procurar por novas soluções.

A funcionalidade de gerar aleatoriamente o problema apresenta a interação com o utilizador bastante semelhante à das figuras em cima, perguntando apenas ao mesmo pelo valor do somatório das colunas. O utilizador pode, ainda, sempre que lhe apetecer, ligar ou desligar as estatísticas a serem imprimidas no ecrã (a partir do menu inicial).

5 Resultados

O utilizador pode verificar os resultados em *real-time*, devolvidos e imprimidos no ecrã pelos predicados *print_time* e *fd_statistics*, podendo ligar ou desligar essa funcionalidade sempre que lhe aprouver. No caso do nosso projeto, podendo ser descrito como um problema de decisão, o tempo necessário para criar uma solução é praticamente automático. A única diferença é mesmo ao nível de *prunings*, *created constraints*, *backtracks* e outras estatísticas devolvidas pelo predicado *fd_statistics*, que serão tanto maiores quanto maior for o número atribuído pelo utilizador para a soma das colunas (ainda que essa diferença seja impercetível em tempo real).

Soma	Solution Time	Backtracks	Created Constraints
12	0.0s	1429	53
19	0.0s	5244	73
23	0.0s	6322	82
27	0.0s	8237	91
39	0.1s	26504	214

Table 1. Tabela de resultados para alguns valores atribuídos pelo utilizador, a partir de um tabuleiro gerado aleatoriamente, demonstrando a ideia geral de que o algoritmo vai demorando mais a encontrar uma solução válida com o aumento da soma.

6 Conclusões e considerações finais

A principal conclusão a retirar deste segundo projeto de Programação em Lógica é a de que a linguagem *Prolog*, mais especificamente referindo-se aos seus módulos de resolução com restrições, é bastante poderosa para resolver uma ampla variedade de questões de decisão/otimização. Uma vez entendido o funcionamento por de trás de variáveis de decisão, formas de restringir o domínio dessas variáveis, a maneira como o *labeling* trabalha, bem como aprender e enquadrar todo o potencial dos predicados existentes na biblioteca 'clpfd', serviu para o nosso grupo ficar bastante à vontade nesta temática, enriquecendo o seu conhecimento deste paradigma de programação.

A solução implementada pelo nosso grupo correspondeu bastante às expetativas e foi de encontro àquilo que era pedido. Ao conseguirmos expandir do caso específico do *Pinwheel Puzzle* para um caso mais abrangente que não só gerasse um problema aleatório, como também o resolvesse (de todas as maneiras possíveis), foi a grande vitória deste trabalho. Perspetivas de trabalho futuro, seriam, por exemplo, fazer com que o utilizador pudesse, além de escolher o número que é suposto dar no somatório de cada coluna, também o tamanho do respetivo tabuleiro (linhas / colunas), ou mesmo os valores a serem usados (e a solução a admitir se seria possível, ou não, resolver o problema com esses valores). Não obstante, o nosso grupo encontra-se completamente satisfeito com a resolução deste trabalho, e sente que não só atingiu, como ainda superou, os objetivos propostos.

7 Bibliografia

[1] Página do Pinwheel Puzzle (com regras, vídeos, etc.), vários autores, http://inveniotoys.com/products/pinwheel-wooden-brain-teaser-puzzle-2-puzzles-in-1

[2] Pierre Deransart, AbdelAli Ed-Dbali, Laurent Cervoni (autores), Prolog: The
Standard: Reference Manual, Springer, 2007.

[3] Ehud Sterling Leon, Shapiro (autor), The Art of Prolog, Second Edition: Advanced Programming Techniques (Logic Programming), 1994.

[5] Slides da disciplina sobre PLR, Henrique Lopes Cardoso (regente), http://moodle.up.pt/course/view.php?id=1028, 2015, consultado em Dezembro de 2015.

[5] Conjunto de dúvidas no StackOverflow sobre Prolog (principalmente sobre listas e restrições), http://stackoverflow.com/questions/tagged/prolog

[6] Vários autores, The Prolog Library - SICStus Prolog, https://sicstus.sics.se/sicstus/docs/4.3.0/html/sicstus/The-Prolog-Library.html, documentação dos predicados disponíveis nas várias bibliotecas do SICStus Prolog.

ANEXO A – Código desenvolvido

O trabalho desenvolvido foi dividido em três ficheiros diferentes, num total de 364 linhas de código.

Ficheiro pinwheel.pl:

```
:- use_module(library(clpfd)).
:- use_module(library(random)).
:- use_module(library(lists)).
:- use_module(library(samsort)).
:- use_module(library(system)).
:- include('utilities.pl').
:- include('randomboard.pl').

% Predicado que inicia a aplicação com as estatisticas desligadas.
% Disponibiliza ao utilizador as opções existentes.
start :- starting(0).

starting(Type) :-
        write('\33\[2J'),
        nl, write('\t\t+ - + - + - + - + - + - + - + - + - + - + - +'), nl, nl,
        write('\t\t- - >   P I N W H E E L   P U Z Z L E   < - -'), nl, nl,
        write('\t\t+ - + - + - + - + - + - + - + - + - + - + - + - +'), nl, nl,
        write('Desenvolvido por Pedro Fraga e Pedro Martins, no ambito de Programacao em Logica.'), nl,
        write('Escolha por favor o que deseja fazer:'), nl, nl,
        write('1 - Gerar um problema no modo "soma 20".'), nl,
        write('2 - Gerar um problema no modo "soma 15".'), nl,
        write('3 - Gerar um problema aleatorio com soma definida.'), nl,
        statisticsOn(Type),
        write('5 - Sair do programa.'), nl, nl,
        repeat,
        write('Opcao (entre 1 e 5) = '),
        read(Option), get_char(_),
        selectCheck(Type, Option).

statisticsOn(0) :-      write('4 - Ligar as estatisticas.'), nl.
statisticsOn(1) :-      write('4 - Desligar as estatisticas.'), nl.

selectCheck(0, Option) :-
        checkOffOption(Option).

selectCheck(1, Option) :-
        checkONOption(Option).
```

```
    % as varias listas para iniciar uma versao do problema.
    firstList([3, 4, 5, 1, 10, 8, 6, 5, 3]).
    secondList([5, 4, 6, 4, 6, 6, 4, 5, 5]).
    thirdList([5, 7, 8, 4, 5, 3, 2, 6, 7]).
    fourthList([8, 5, 6, 2, 1, 4, 6, 7, 4]).

    % checkOffOption/1 pode ter um valor entre 1 e 5. Todos os outros são opções inválidas.
    checkOffOption(1) :- hardMode(0, _).
    checkOffOption(2) :- easyMode(0,_).
    checkOffOption(3) :- randomMode(0,_).
    checkOffOption(4) :- starting(1).
    checkOffOption(5) :- abruptExit.
    checkOffOption(_)
            :- nl, write('!!! ERRO !!! Opcao invalida, deve ser um numero entre 1 e 5. Por favor escolha novamente!'), nl, nl, false.

    % checkONOption/1 pode ter um valor entre 1 e 5. Todos os outros são opções inválidas.
    checkONOption(1) :- hardMode(1,_).
    checkONOption(2) :- easyMode(1,_).
    checkONOption(3) :- randomMode(1,_).
    checkONOption(4) :- starting(0).
    checkONOption(5) :- abruptExit.
    checkONOption(_)
            :- nl, write('!!! ERRO !!! Opcao invalida, deve ser um numero entre 1 e 5. Por favor escolha novamente!'), nl, nl, false.

    permutate([], []).
    permutate([Line|Board], [ResultLine|Result]):-
            length(Line, N),
            length(ResultLine, N),
            length(P, N),
            sorting(ResultLine, P, Line),
            permutate(Board, Result).

    sumBoard([], _).
    sumBoard([Column|Columns], Sum):-
            sum(Column, #=, Sum),
            sumBoard(Columns, Sum).

    sortBoard([], []).
    sortBoard([L|B], [SL|SB]):- samsort(L, SL),
      sortBoard(B, SB).
```

```
    solver(Board, Result, Sum):-
        sortBoard(Board, Sorted),
        permutate(Sorted, Result),
        transpose(Result, Columns),
        sumBoard(Columns, Sum),
        flattenList(Result, Results),
        labeling([], Results).

    flattenList([],[]).
    flattenList([L1|Ls], Lf):- is_list(L1), flattenList(L1, L2), append(L2, Ld, Lf),
flattenList(Ls, Ld).
    flattenList([L1|Ls], [L1|Lf]):- \+is_list(L1), flattenList(Ls, Lf).

    easyMode(Statistics, Result) :-
            write('\33\[2J'),
            nl, write('\t\t + - + - + - + - + - + - + - + - + - + - + - + - + - +'), nl, nl,
            write('\t\t - - >  P I N W H E E L - S O M A   1 5  < - -'), nl, nl,
            write('\t\t + - + - + - + - + - + - + - + - + - + - + - + - + - + - +'), nl, nl,
            nl, write('-----------------------------------------------------------------------------------------'),
nl,
            write('\t\t\t   O problema foi gerado. '), nl,
            write('O objetivo consiste em organizar cada linha de forma a que o somatorio
das colunas de 15. '),
            nl, write('-----------------------------------------------------------------------------------------'),
nl, nl,
            secondList(L1),
            thirdList(L2),
            fourthList(L3),
            shuffleList(L1, RL1),
            shuffleList(L2, RL2),
            shuffleList(L3, RL3),
            drawBoard([RL1, RL2, RL3]), firstUserChoice(Choice),
            easyModeNext(Statistics, Choice, Result, RL1, RL2, RL3).

    hardMode(Statistics, Result) :-
            write('\33\[2J'),
            nl, write('\t\t + - + - + - + - + - + - + - + - + - + - + - + - + - +'), nl, nl,
            write('\t\t - - >  P I N W H E E L - S O M A   2 0  < - -'), nl, nl,
            write('\t\t + - + - + - + - + - + - + - + - + - + - + - + - + - + - +'), nl,
            nl, write('-----------------------------------------------------------------------------------------'),
nl,
            write('\t\t\t   O problema foi gerado. '), nl,
            write('O objetivo consiste em organizar cada linha de forma a que o somatorio
das colunas de 20. '),
```

```
        nl, write('-------------------------------------------------------------------------------------'),
nl, nl,
        firstList(L1),
        secondList(L2),
        thirdList(L3),
        fourthList(L4),
        shuffleList(L1, RL1),
        shuffleList(L2, RL2),
        shuffleList(L3, RL3),
        shuffleList(L4, RL4),
        drawBoard([RL1, RL2, RL3, RL4]), firstUserChoice(Choice),
        hardModeNext(Statistics, Choice, Result, RL1, RL2, RL3, RL4).

hardModeNext(1, 1, Result, RL1, RL2, RL3, RL4) :-
        reset_timer,
        solver([RL1, RL2, RL3, RL4], Result, 20), nl, print_time, fd_statistics,
        nl, drawBoard(Result), secondUserChoice, fail.
hardModeNext(0, 1, Result, RL1, RL2, RL3, RL4) :-
        solver([RL1, RL2, RL3, RL4], Result, 20),
        nl, drawBoard(Result), secondUserChoice, fail.
hardModeNext(Statistics, 2, Result, _, _, _, _) :-
        hardMode(Statistics, Result).

easyModeNext(1, 1, Result, RL1, RL2, RL3) :-
        reset_timer,
        solver([RL1, RL2, RL3], Result, 15), nl, print_time, fd_statistics,
        nl, drawBoard(Result), secondUserChoice, fail.
easyModeNext(0, 1, Result, RL1, RL2, RL3) :-
        solver([RL1, RL2, RL3], Result, 15),
        nl, drawBoard(Result), secondUserChoice, fail.
easyModeNext(Statistics, 2, Result, _, _, _) :-
        easyMode(Statistics, Result).

shuffleList(A, B) :- random_permutation(A, B).

sumBigBoard(B, Sum) :-
        transpose(B, NewB),
        sumColumns(NewB, [], Sum).

sumColumns([], B, B).
sumColumns([X | Xs], B, Final) :-
        sumLine(X, Sum),
        append(B, [Sum], NewList),
        sumColumns(Xs, NewList, Final).
```

```
sumLine([], 0).
sumLine([X | Xs], Sum) :-
        sumLine(Xs, Sum1),
        Sum is X + Sum1.

% Predicado que desenha o puzzle para interação com o utilizador.
% Faz uso de um outro predicado auxiliar para desenhar cada linha individualmente.
drawBoard([], S, SumList):-
        write('\t    +'), drawTop(S),
        write('\t    '), drawEnd(S), nl,
        write('\t     '), drawColumnSum(SumList), nl , nl, !.

drawBoard([X | Xs], S, SumList) :-
        write('\t    +'), drawTop(S),
        write('\t    |'), drawMiddle(S),
        write('\t    |'), drawLine(X), nl,
        write('\t    |'), drawMiddle(S),
        drawBoard(Xs, S, SumList), !.

drawBoard([X | Xs]) :-
        sumBigBoard([X | Xs], SumList),
        getSize([X | Xs], S),
        write('\t    +'), drawTop(S),
        write('\t    |'), drawMiddle(S),
        write('\t    |'), drawLine(X), nl,
        write('\t    |'), drawMiddle(S),
        drawBoard(Xs, S, SumList), !.

% Predicado para desenhar uma linha do tabuleiro (convertendo cada valor individual).
% Condição de terminação: lista vazia (percorreu a linha até ao fim).
drawLine([]).
drawLine([X | Xs]) :-
        convertValue(X), write('|'), drawLine(Xs), !.

drawColumnSum([]).
drawColumnSum([X | Xs]) :-
        convertValue(X), write(' '), drawColumnSum(Xs), !.

convertValue(X) :-
        X > 9, write(' '), write(X), write('  '), !.
convertValue(X) :-
        write('  '), write(X), write('  '), !.

drawTop(0) :- nl, !.
```

```
drawTop(S) :-
        write('-----+'),
        S1 is S - 1, drawTop(S1), !.

drawMiddle(0) :- nl, !.
drawMiddle(S) :-
        write('     |'),
        S1 is S - 1, drawMiddle(S1), !.

drawMiddlePlus(0) :- nl, !.
drawMiddlePlus(S) :-
        write('     +'),
        S1 is S - 1, drawMiddlePlus(S1), !.

drawEnd(0) :- nl, !.
drawEnd(S) :-
        write('   =  '),
        S1 is S - 1, drawEnd(S1), !.

getSize([H | _], S) :-
        length(H, S), !.
```

Ficheiro randomboard.pl:

```
randomMode(Statistics, _) :-
        write('\33\[2J'),
        nl, write('\t\t+ - + - + - + - + - + - + - + - + - + - + - + - +'), nl, nl,
        write('\t\t- - >   P I N W H E E L   P U Z Z L E   < - -'), nl, nl,
        write('\t\t+ - + - + - + - + - + - + - + - + - + - + - + - +'), nl, nl,
        write('Desenvolvido por Pedro Fraga e Pedro Martins, no ambito de
Programacao em Logica.'), nl,
        write('Escolha por favor o valor da soma resultante entre colunas:'), nl, nl,
        repeat,
        write('Valor (entre 10 e 40) = '),
        read(Option), get_char(_),
        Option >= 10, Option =< 40,
        printGeneratingRandomBoard(Statistics, Option).

printGeneratingRandomBoard(Statistics, Sum) :-
        write('\33\[2J'),
        nl,
        nl, write('\t\t+ - + - + - + - + - + - + - + - + - + - + - + - +'), nl, nl,
        write('\t\t- - >   P I N W H E E L   P U Z Z L E   < - -'), nl, nl,
        write('\t\t+ - + - + - + - + - + - + - + - + - + - + - + - +'), nl, nl,
```

```
        write('Desenvolvido por Pedro Fraga e Pedro Martins, no ambito de
Programacao em Logica.'), nl,
        nl,
        nl,
        write('A gerar um puzzle aleatorio cuja soma entre colunas resulte em '),
        write(Sum),
        write('...'),
        nl,
        nl,
        write('Por favor aguarde...'),
        nl,
        nl,
        nl,
        initializeRandomSeed,
        generateRandomBoard(Statistics, Sum).

generateRandomBoard(Statistics, Sum) :-
        random(4, 10, NCol),
        random(4, 10, NLin),
        createBoard(Board, NCol, NLin, 0, FinalEqualNumbers),
        transpose(Board, Columns),
    sumBoard(Columns, Sum),
    flattenList(Board, Results),
    labeling( [maximize(FinalEqualNumbers), time_out(3, Flag)] , Results),
        drawRandomBoard(Statistics, Sum, Board, Flag).

drawRandomBoard(Statistics, Sum, _, time_out) :-
        generateRandomBoard(Statistics, Sum).

drawRandomBoard(Statistics, Sum, Board, _)      :-
        shuffleBoard(Board, Result),
        drawBoard(Result),
        firstUserChoice(Option),
        finalRandom(Statistics, Option, Sum, Result).

finalRandom(1, 1, Sum, Board) :-
        reset_timer,
        solver(Board, Result, Sum), nl, print_time, fd_statistics,
        nl, drawBoard(Result), secondUserChoice, fail.

finalRandom(0, 1, Sum, Board) :-
        solver(Board, Result, Sum), nl,
        nl, drawBoard(Result), secondUserChoice, fail.

finalRandom(_, 1, _, _) :-
```

```
        nl, nl, nl,
        write('Nao ha mais solucoes disponiveis...'),
        nl, nl,
        abruptExit.

finalRandom(Statistics, 2, Sum, _) :-
        generateRandomBoard(Statistics, Sum).

finalRandom(_, 3, _, _).

shuffleBoard( [] , _).

shuffleBoard( [L1 | LRest] , [SL1 | SLRest]) :-
        shuffleList(L1, SL1),
        shuffleBoard(LRest, SLRest).

createBoard(_, _, 0, N, Final) :-
        Final #= N.

createBoard([Lin | Rest], NCol, NLin, N, Final) :-
        length(Lin, NCol),
        domain(Lin, 0, 10),
        nvalue(Number, Lin),
        N1 #= Number + N,
        NLin1 is NLin - 1,
        createBoard(Rest, NCol, NLin1, N1, Final).
```

Ficheiro utilities.pl:

```
firstUserChoice(Option) :-
        nl, write('Escolha por favor o que deseja fazer:'), nl, nl,
        write('1 - Encontrar uma solucao para o problema.'), nl,
        write('2 - Gerar uma nova versao do problema.'), nl,
        write('3 - Voltar ao menu inicial.'), nl, nl,
        repeat,
        write('Opcao (entre 1 e 3) = '),
        read(Option), checkFirstUserChoice(Option).

checkFirstUserChoice(1).
checkFirstUserChoice(2).
checkFirstUserChoice(3) :- start.
checkFirstUserChoice(_)
        :- nl, write('!!! ERRO !!! Opcao invalida, deve ser um numero entre 1 e 3. Por
favor escolha novamente!'), nl, nl, false.
```

```
secondUserChoice :-
        nl, write('Esta solucao agrada-lhe?:'), nl, nl,
        write('1 - Sim!'), nl,
        write('2 - Nao, desejo outra solucao.'), nl, nl,
        repeat,
        write('Opcao (entre 1 e 2) = '),
        read(Option), !, checkSecondUserChoice(Option).

checkSecondUserChoice(1) :- niceExit.
checkSecondUserChoice(2) :- reset_timer, fail.
checkSecondUserChoice(A)
        :- A \= 1, A \= 2, nl, write('!!! ERRO !!! Opcao invalida, deve ser um numero
entre 1 e 2. Por favor escolha novamente!'), nl, nl, secondUserChoice.

% Predicado para sair do programa, imprimindo a respetiva mensagem no ecrã.
abruptExit :-
        nl, write('-----------------------------------------------------------------------'), nl,
        write('O programa vai agora fechar. Para abri-lo novamente escreva "start."'),
        nl, write('-----------------------------------------------------------------------'),nl, nl, abort.

niceExit :-
        nl, write('-----------------------------------------------------------------------------'), nl,
        write('Obrigado por usar o nosso programa! Para abri-lo novamente escreva
"start."'),
        nl,     write('-----------------------------------------------------------------------------'),nl,     nl,
abort.

initializeRandomSeed:-
        now(Usec), Seed is Usec mod 30269,
        getrand(random(X, Y, Z, _)),
        setrand(random(Seed, X, Y, Z)), !.

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.
```