Pinwheel Puzzle

Resolução de um Problema de Decisão usando Programação em Lógica com restrições

FEUP-PLOG, Turma 3MIEIC1, Grupo Pinwheel Puzzle_2

Miguel Pereira e Pedro Silva

Resumo. O artigo tem como objetivo complementar o segundo projeto realizado para a Unidade Curricular de Programação em Lógica. O programa é capaz de resolver qualquer tabuleiro do jogo Pinwheel Puzzle.

1 Introdução

O objetivo deste trabalho é implementar a resolução de um problema de decisão ou otimização utilizando Programação em Lógica com restrições.

Este grupo escolheu o Pinwheel Puzzle, um problema de decisão. Neste jogo, o jogador tem de ordernar um conjunto de peças num tabuleiro circular de forma a cumprir determinados requisitos.

Este artigo irá explicar mais detalhadamente o puzzle escolhido e o seu objetivo; abordagem do grupo na implementação de uma resolução capaz de gerar todas as soluções possíveis para um tabuleiro; a explicação da forma usada para representar uma solução; os resultados obtidos com diferentes complexidades e a conclusão do artigo.

2 Descrição do problema

O puzzle Pinwheel Puzzle é jogado num tabuleiro circular dividido em várias regiões circulares com peças próprias com um número embutido que só podem ser usadas na regiões correspondentes. Para além disso, é ainda necessário considerer as várias "fatias" em que o tabuleiro está dividido. O objetivo do puzzle é dispor as peças de tal forma que a soma de cada "fatia" seja igual a um determinado número.

No jogo original existem duas dificuldades: uma onde se deve ignorar a região circular mais exterior e e ordenar as peças de modo a que a soma de cada "fatia" seja 15 e outra onde a soma de cada "fatia" seja 20, tendo em conta a região circular mais exterior.

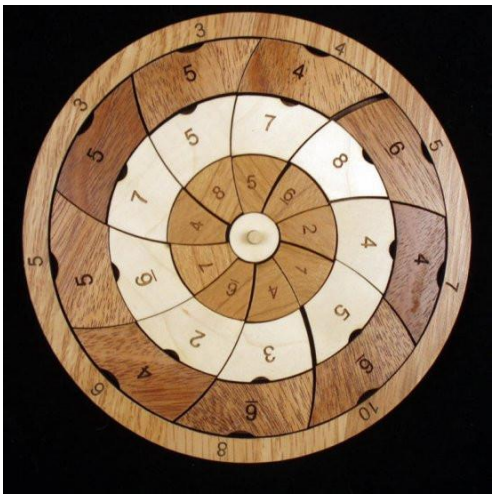

Fig. 1. Exemplo de um tabuleiro[1]

[1] O tabuleiro representado tem um erro: para a resolução do puzzle ser possível de acordo com as regras originais, na região circular mais exterior, onde está um "7", deveria estar um "1".

3 Abordagem

O tabuleiro de jogo é representado por uma lista de listas, onde cada lista representa uma região circular. As listas contêm os valores das peças contidas nas regiões circulares respetivas.

3.1 Variáveis de Decisão

A solução pretendida pode ser representada por uma lista de listas. O número de listas desta lista depende do número de regiões circulares existentes e o número de elementos de cada uma das listas depende do número de "fatias".

O que é passado à função de *labeling* é uma versão *non-nested* de uma permutação do tabuleiro original, sendo que o domínio será do número minímo desta lista até ao número máximo.

3.2 Restrições

Existe apenas uma restrição: a soma de cada "fatia" tem de ser ***Sum***. O predicado referente a esta restrição é o seguinte:

```
checkSum(Columns, Sum)
```

4 Visualização da solução

O tabuleiro não é representado em formato circular por uma questão de facilidade na visualização, mas sim como um conjunto de colunas que representam as fatias.

```
 ___   ___   ___   ___   ___   ___   ___   ___   ___
| 1 | | 3 | | 3 | | 4 | | 5 | | 5 | | 6 | | 8 | | 10 |
 ___   ___   ___   ___   ___   ___   ___   ___   ___
| 4 | | 4 | | 4 | | 5 | | 5 | | 5 | | 6 | | 6 | | 6 |
 ___   ___   ___   ___   ___   ___   ___   ___   ___
| 7 | | 6 | | 7 | | 5 | | 5 | | 8 | | 4 | | 2 | | 3 |
 ___   ___   ___   ___   ___   ___   ___   ___   ___
| 8 | | 7 | | 6 | | 6 | | 5 | | 2 | | 4 | | 4 | | 1 |
 ___   ___   ___   ___   ___   ___   ___   ___   ___
```

Fig. 2. Exemplo de representação de uma solução

O predicado responsável pela impressão do tabuleiro é o predicado

```
printResult(Result)
```

que recebe uma lista de listas e imprime o tabuleiro linha a linha.

5 Resultados

Os testes efetuados não foram suficientemente extensos, portanto os resultados obtidos não são muito conclusivos. Nas medições que se fizeram, as soluções foram sempre encontradas em menos de 0.00 segundos. Será de esperar que o tempo para encontrar uma solução aumente com o aumento do tabuleiro.

6 Conclusões

O grupo conseguiu perceber e interpretar facilmente o puzzle escolhido. Inicialmente, tivemos alguma dificuldade a decidir qual a melhor maneira de resolver o problema proposto, mas assim que se decidiu que solução implementar, a sua tradução para código foi relativamente rápida e sem grandes dificuldades, sendo que a maior dificuldade foi a implementação de uma função que transformasse uma lista *nested* numa lista *non-nested*. A solução implementada não tem limitações, uma vez que é capaz de resolver qualquer tabuleiro dado.

Bibliografia

1. Pinwheel Puzzle, http://www.creativecrafthouse.com/index.php?main_page=product_info&cPath=96&products_id=871
2. SWI-Prolog, http://www.swi-prolog.org/

Anexo

PinwheelPuzzle.pl

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

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%         INTERFACE       %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pinwheelPuzzle:-
  clrscr,
  write('   :: Pinwheel Puzzle ::   '), nl,
  write('                            '), nl,
  write('   1. Solve in easy mode    '), nl,
  write('   2. Solve in hard mode    '), nl,
  write('   3. Exit                  '), nl,
  write('                            '), nl,
  write('   > '),
  getChar(OP), (
    OP = '1' -> clrscr, getEasyModeBoard(Board),
solveAndShow(Board, 15), pinwheelPuzzle;
    OP = '2' -> clrscr, getHardModeBoard(Board),
solveAndShow(Board, 20), pinwheelPuzzle;
    OP = '3';
    pinwheelPuzzle).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%          SOLVER         %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

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

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

```
sortBoard([], []).
sortBoard([Line | Board], [SortedLine | SortedBoard]):-
  samsort(Line, SortedLine),
  sortBoard(Board, SortedBoard).

flattenList([], []).
flattenList([Line | List], Result):-
  is_list(Line), flattenList(Line, Result2), append(Re-
sult2, Tmp, Result), flattenList(List, Tmp).
flattenList([Line | List], [Line | Result]):-
  \+is_list(Line), flattenList(List, Result).

solvePinwheel(Board, Result, Sum):-
  sortBoard(Board, SortedBoard),
  permutations(SortedBoard, Result),
  transpose(Result, Columns),
  checkSum(Columns, Sum),
  flattenList(Result, Results),
  labeling([], Results).

 solveAndShow(Board, Sum):-
  solvePinwheel(Board, Result, Sum),
  nl, write('   Press enter to show next solution or
write "e" to exit. '),
  get_char(C),
  ((C = 'e'; C = 'E') -> get_char(_);
  clrscr, printResult(Result)), !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%         PRINTER        %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

printResult(Result):-
  write('   '),
  printSeparator(0, 9), nl,
  addElem(Result, 0, 9), !.

addElem(_, S, S).
addElem([H|T], I, S):-
  I1 is I + 1,
  write('  '),
  createLine(H, 0, S),
  nl, write('   '), printSeparator(0, S), nl, !,
  addElem(T, I1, S).
```

```
createLine(_, S, S).
createLine([A|B], I, S):-
  I1 is I + 1,
  write('| '),
  write(A),
  write(' | '),
  createLine(B, I1, S).

printSeparator(S, S).
printSeparator(0, S):-
  write('--- '),
  printSeparator(1, S), !.
printSeparator(A, S):-
  A1 is A + 1,
  write('  --- '),
  printSeparator(A1, S), !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%          BOARDS          %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getEasyModeBoard(Board):-
  Board = [[5, 4, 6, 4, 6, 6, 4, 5, 5],
      [5, 7, 8, 4, 5, 3, 2, 6, 7],
      [8, 5, 6, 2, 1, 4, 6, 7, 4]].

getHardModeBoard(Board):-
  Board = [[3, 4, 5, 1, 10, 8, 6, 5, 3],
      [5, 4, 6, 4, 6, 6, 4, 5, 5],
      [5, 7, 8, 4, 5, 3, 2, 6, 7],
      [8, 5, 6, 2, 1, 4, 6, 7, 4]].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%         UTILITIES        %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getChar(C):-
  get_char(C),
  get_char(_).

clrscr:-
  clrscr(70), !.
clrscr(0).
clrscr(N):-
```

```
nl,
N1 is N - 1,
clrscr(N1).
```