Magic Snail

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo TP4_3:

Manuel Curral,
Nelson Costa

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

23 de dezembro de 2017

Resumo

Pretendeu-se dar solução, em PLR, a um puzzle, nomeadamente, *Magic Snail,* com três principais restrições, visando atingir resultados eficazes e eficientes, ao fazer variar a complexidade do problema. Este relatório prende-se com a exposição introdutória e evolutiva do desenvolvimento teórico-prático do problema e da análise interpretativa dos dados obtidos.

Conteúdo

Introdução

Este trabalho insere-se no segundo momento avaliativo da componente teórico-prática da unidade curricular de Programação em Lógica do 3º ano do Mestrado Integrado em Engenharia Informática e Computação da FEUP. De entre puzzles, como o *Bosnian Snake, Fence, etc*, e problemas de otimização, como distribuição do corpo docente ou redistribuição do público, escolhemos o puzzle *Magic Snail*.

Neste artigo, abordaremos as principais características do problema desenvolvido, assim como a nossa abordagem de solução escolhida e os resultados obtidos, com a respetiva análise.

Puzzle *Magic Snail*:

Este puzzle pode considerar-se agrupado nos quebra-cabeça do género do *Sudoku*, em que dada uma chave, que pode ser constituída por letras ou números, e um tabuleiro quadrado, o utilizador terá que chegar a uma solução.

A solução deste jogo é obtida satisfazendo três restrições principais: cada elemento da chave tem que aparecer uma e apenas uma única vez em cada linha; cada elemento da chave tem que aparecer uma e apenas uma única vez em cada coluna; o tabuleiro deve ser lido na forma de espiral desde o canto superior esquerdo até ao seu centro, e o seu conteúdo ao longo dessa espiral deve conter sequências da chave, pela ordem certa.

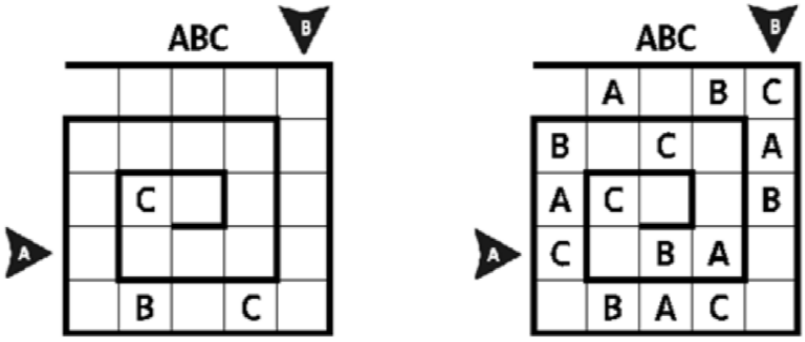


Fig1: Demonstração da lógica do puzzle.

Abordagem de resolução

Variáveis:

A solução do puzzle é representada internamente por uma lista simples. Como tal, as variáveis de decisão constituem uma lista simples que contém os elementos da chave ordenados, sendo o seu domínio de 0 ao tamanho da chave.

```
  Spiral_len is BoardLength*BoardLength,
  length(Spiral, Spiral_len),
%domain
  domain(Spiral, 0, KeyLength),
```

Fig2: Instanciação da lista-solução e do domínio do problema.

Restrições:

1. Cada linha só pode ter uma ocorrência de cada elemento da chave.:

```
checkRow([],_,_):-!.
checkRow(Spiral, BoardLength, KeyLength):-
    splitByLength(BoardLength, Spiral, SplitElems), %gets a list of the first row elems
    append(SplitElems,RestElems,Spiral), %to get the rest of the list
    checkRow(RestElems,BoardLength, KeyLength),

    getKeyIndexedList(KeyLength,IndexedList), %gets an indexed list from 1 to the key's length
    setEachIndexCount(IndexedList, SplitElems, 1). %makes sure that there's only 1 of each index on the split part
```

Fig3: predicados que garantem a restrição de linha.

2. Cada coluna só pode ter uma ocorrência de cada elemento da chave.

```
%gets a version of the list in which the order of the elements is like they've written there by columns
checkCol([],_,_,_):-!.
checkCol(_,BoardLength,_, N):-
    N > BoardLength,!.
checkCol(Spiral, BoardLength, KeyLength, N1):-
    Next is N1+1,
    checkCol(Spiral, BoardLength, KeyLength, Next),
    getColElems(N1, BoardLength, Spiral, ColElems),
    checkRow(ColElems, BoardLength, KeyLength).
```

Fig4: predicados que garantem a restrição de coluna.

3. A chave deve surgir ao longo lista na sua sequência original, as vezes necessárias até ao fim do tabuleiro.

```
checkSequence(List, BoardLength):-
    setSpiralPath(BoardLength, Path), %gets a list in which the elements are the direction of the spiral
    setSpiral(List,Path, BoardLength,Spiral,0,0), %sets the indexes in the list as if it was a matrix
    setSequence(Spiral,BoardLength,1,0). %sets the constraints of the sequence
```

Fig5: predicados que garantem a restrição de sequência.

Estrutura de pesquisa:

A estrutura de labeling utilizada foi a fcc, ou seja, as variáveis são dependentes da ordem do domínio e número de restrições. Como neste puzzle, to domínio é único, o critério de organização foi apenas as restrições.

```
%result
  labeling([ffc], Spiral).
```

Fig6: predicado de labelling para encontrar a solução do problema.

Visualização da solução

```
?-magicSnail(abc,5).
+-----+-----+-----+-----+-----+
   a     b                 c  |
+-----+-----+-----+-----+     +
|  c           a     b  |     |
+     +-----+-----+     +     +
|  b  |           |  c  |  a  |
+     +     +-----+     +     +
|     |  c     b     a  |     |
+     +-----+-----+-----+     +
|        a     c           b  |
+-----+-----+-----+-----+-----+
Execution took 150 ms.
```

Fig7: Solução do puzzle em modo de texto.

Para apresentar a solução num modo visual ao utilizador, utilizaram-se vários predicados lógicos e outros de desenho do "tabuleiro" propriamente dito.

A nível de predicados lógicos, a abordagem utilizada foi: através do predicado ***setPuzzle(Spiral, BoardLength, KeyLength)***, que, devolve uma lista com os índices de posição preenchidos que serão associados aos índices de posição da chave. Esta lista é, então utilizada pelo predicado ***placeKeys(Spiral, Result, KeyList),*** que preenche a lista *Result* com os valores dos elementos da chave respetivos. Esta lista é processada, por fim, por uma função auxiliar, ***listToMatrix(Result, BoardLength, Matrix)***, que a converte numa matriz, ***Matrix***, que será utilizada no sentido de ser desenhada na linha de comandos.

Quanto aos predicados de desenho propriamente ditos, foi utilizada a estratégia de dividir os elementos da matriz como sendo *empty* (vazios), *vertical_line* (linhas verticais), *horizontal_line* (linhas horizontais), *vertical_empty* e *horizontal_empty* (linhas

horizontais ou verticais que devem estar vazias). Tudo isto para conseguir desenhar apenas os limites da espiral por onde a lógica do jogo se processa.

```
setPuzzle(Spiral, BoardLength, KeyLength):-
    Spiral_len is BoardLength*BoardLength,
    length(Spiral, Spiral_len),
  %domain
    domain(Spiral, 0, KeyLength),
  %constraints
    checkRow(Spiral, BoardLength, KeyLength), %making sure each element of the key shows up only once every row
    checkCol(Spiral, BoardLength, KeyLength,1), %making sure each element of the key shows up only once every column
    checkSequence(Spiral, BoardLength), %making sure the sequence of the elements in the spiral is right
  %result
    labeling([ffc], Spiral).
```

Fig8: predicado que devolve a lista indexada.

```
%fills a list with the direction of the next 'move', considering the board's length
setSpiralPath(1,[]):-!.
setSpiralPath(N,Path):-
    N2 is N-2,
    setSpiralPath(N2, PathTail),
    N1 is N-1,
    setListAt(Right, d, N1),
    setListAt(Down, s, N1),
    setListAt(Left, a, N1),
    setListAt(Up, w, N2),

    append(Right, Down, RD),
    append(RD, Left, RDL),
    append(RDL, Up, RDLU),
    append(RDLU, [d], TmpPath),

    append(TmpPath, PathTail, Path).

setSpiral(List, [], BoardLength, Spiral ,Row,Column):-
    Index is Row*BoardLength + Column,
    nth0(Index, List, Elem),
    Spiral = [Elem].

setSpiral(List, [Direction|PathTail], BoardLength, Spiral, Row,Column):-
    switch(Direction, [
        d:(NewCol is Column+1, NewRow is Row),
        a:(NewCol is Column-1,NewRow is Row),
        s:(NewCol is Column, NewRow is Row+1),
        w:(NewCol is Column, NewRow is Row-1)]),
    setSpiral(List, PathTail, BoardLength, SpiralTail,NewRow,NewCol),
    Index is Row*BoardLength + Column,
    nth0(Index, List, Elem),
    Spiral = [Elem | SpiralTail].
```

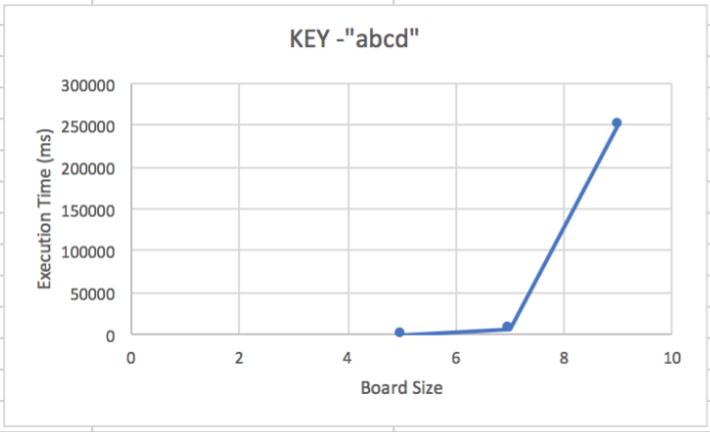
Fig9: predicados de indexação da lista.

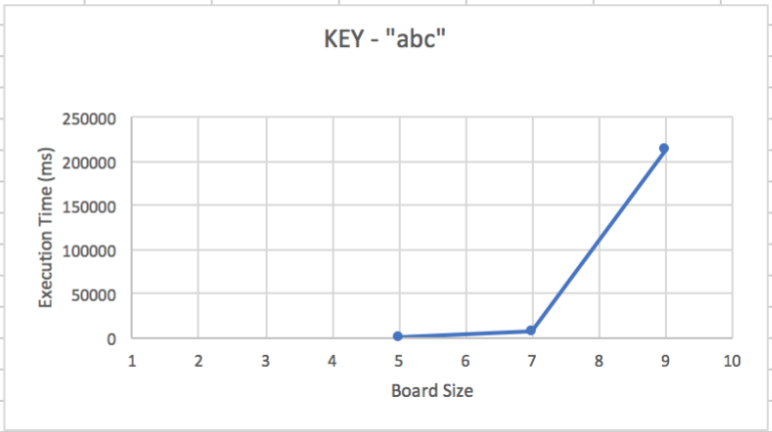
Resultados

Os resultados obtidos são os seguintes:

KEY	
"abc"	
BoardSize NxN	Exec Time(ms)
5	160
7	5960
9	212320

KEY	
"abcd"	
BoardSize NxN	Exec Time(ms)
5	200
7	7170
9	250900





Utilizou-se várias chaves e vários tamanhos de tabuleiro. Os predicados correm com sucesso para todas as chaves de 3 elementos e para tabuleiros de tamanho superior a 9 o programa demora bastante tempo a correr.

Conclusões

Com este projeto, conseguiu-se, de facto, consolidar os conhecimentos teóricos e práticos inerentes a esta unidade curricular assim como desenvolver capacidades de raciocínio lógico perante a resolução deste problema.

Conclui-se que a solução apresentada, embora cumpra o pretendido, possa não ser a mais eficiente. Uma melhor e mais centrada implementação das restrições seria, porventura, um dos aspetos a melhorar, de modo a conseguir registar melhores resultados. Contudo, com a falta de tempo, não conseguimos ir mais além.

Referências

http://www.swi-prolog.org;
Sterling, Leon; The Art of Prolog. ISBN: 0-262-69163-9
Marriot, Kim; Programming with constraints. ISBN: 0-262-13341-5

Anexos

Código Fonte:

main.pl:

```
:- use_module(library(clpfd)).
:- use_module(library(lists)).
:- include('board.pl').
:- include('logic.pl').
:- include('misc.pl').


magicSnail(Key, BoardLength):-
    statistics(walltime, [TimeSinceStart | [TimeSinceLastCall]]),
    atom_chars(Key,KeyList),
    length(KeyList, KeyLength),
    setPuzzle(Spiral,BoardLength,KeyLength),
    placeKeys(Spiral, Result, KeyList),
    listToMatrix(Result, BoardLength, Matrix),
    buildBoard(Matrix, BoardLength, Board),
    displayBoard(Board),
    statistics(walltime, [NewTimeSinceStart | [ExecutionTime]]),
    write('Execution took '), write(ExecutionTime), write(' ms.'), nl.
```

logic.pl:

```
setPuzzle(Spiral, BoardLength, KeyLength):-
    Spiral_len is BoardLength*BoardLength,
    length(Spiral, Spiral_len),
  %domain
    domain(Spiral, 0, KeyLength),
  %constraints
    checkRow(Spiral, BoardLength, KeyLength), %making sure each element of the key shows up only once every row
    checkCol(Spiral, BoardLength, KeyLength,1), %making sure each element of the key shows up only once every column
    checkSequence(Spiral, BoardLength), %making sure the sequence of the elements in the spiral is right
  %result
    labeling([ffc], Spiral).

checkRow([],_,_):-!.
checkRow(Spiral, BoardLength, KeyLength):-
    splitByLength(BoardLength, Spiral, SplitElems), %gets a list of the first row elems
    append(SplitElems,RestElems,Spiral), %to get the rest of the list
    checkRow(RestElems,BoardLength, KeyLength),

    getKeyIndexedList(KeyLength,IndexedList), %gets an indexed list from 1 to the key's length
    setEachIndexCount(IndexedList, SplitElems, 1). %makes sure that there's only 1 of each index on the split part

%gets a version of the list in which the order of the elements is like they've written there by columns
checkCol([],_,_,_):-!.
checkCol(_,BoardLength,_, N):-
    N > BoardLength,!.
checkCol(Spiral, BoardLength, KeyLength, N1):-
    Next is N1+1,
    checkCol(Spiral, BoardLength, KeyLength, Next),
    getColElems(N1, BoardLength, Spiral, ColElems),
    checkRow(ColElems, BoardLength, KeyLength).


checkSequence(List, BoardLength):-
    setSpiralPath(BoardLength, Path), %gets a list in which the elements are the direction of the spiral
    setSpiral(List,Path, BoardLength,Spiral,0,0), %sets the indexes in the list as if it was a matrix
    setSequence(Spiral,BoardLength,1,0). %sets the constraints of the sequence

%fills a list with the direction of the next 'move', considering the board's length
setSpiralPath(1,[]):-!.
setSpiralPath(N,Path):-
    N2 is N-2,
    setSpiralPath(N2, PathTail),
    N1 is N-1,
    setListAt(Right, d, N1),
    setListAt(Down, s, N1),
    setListAt(Left, a, N1),
    setListAt(Up, w, N2),

    append(Right, Down, RD),
    append(RD, Left, RDL),
    append(RDL, Up, RDLU),
    append(RDLU, [d], TmpPath),

    append(TmpPath, PathTail, Path).

setSpiral(List, [], BoardLength, Spiral ,Row,Column):-
    Index is Row*BoardLength + Column,
    nth0(Index, List, Elem),
    Spiral = [Elem].

setSpiral(List, [Direction|PathTail], BoardLength, Spiral, Row,Column):-
    switch(Direction, [
        d:(NewCol is Column+1, NewRow is Row),
        a:(NewCol is Column-1,NewRow is Row),
        s:(NewCol is Column, NewRow is Row+1),
        w:(NewCol is Column, NewRow is Row-1)]),
    setSpiral(List, PathTail, BoardLength, SpiralTail,NewRow,NewCol),
    Index is Row*BoardLength + Column,
    nth0(Index, List, Elem),
    Spiral = [Elem | SpiralTail].
```

```
setSequence(_,BoardLength, N,_):-
    N>BoardLength.
setSequence(Spiral,BoardLength,Counter, LastOccur):-
    keyCounter(Spiral, 1, Counter, Index1),
    keyCounter(Spiral, 2, Counter, Index2),
    keyCounter(Spiral, 3, Counter, Index3),
            LastOccur#<Index1,
            Index1#<Index2,
            Index2#<Index3,
    Next is Counter+1,
    setSequence(Spiral,BoardLength,Next, Index3).

keyCounter(Spiral, Element, Limit, Index) :-
  keyCounterLimit(Spiral,Element, 0, Limit, Index, 0),
  element(Index,Spiral,Element).

keyCounterLimit(_,_,Count,Count,Index,Index).
keyCounterLimit([Head|Tail], Element, Aux, Count, Index, IndexAux) :-
    Head #= Element #<=> Bool,
    Aux2 #= Aux + Bool,
    N_Index is IndexAux + 1,
    keyCounterLimit(Tail, Element, Aux2, Count, Index, N_Index).

setIndexCount(_, [], 0).
setIndexCount(Value, [Head | Tail], Count) :-
    setIndexCount(Value, Tail, Count2),
    Value #= Head #<=> Bool,
    Count #= Count2 + Bool.

setEachIndexCount([],_,_).
setEachIndexCount([Value|Rest], List, Count):-
    setEachIndexCount(Rest, List, Count),
    setIndexCount(Value,List,Count).
```

board.pl

```
meaning(empty, '     '):-!.
meaning(h_line, '------'):-!.
meaning(v_empty, ' '):-!.
meaning(v_line, '|'):-!.
meaning(h_empty, '     '):-!.
meaning(X, Res):-
    atom_concat('  ', X, X1),
    atom_concat(X1, '  ', Res).


buildBoard(Matrix, BoardLength, Board):-
    horizontalLines(BoardLength, HLine),
    verticalLines(BoardLength, VLine),
    buildBoard(Matrix, HLine, VLine, BoardLength, Board).

buildBoard([], [RowHLine], [], _, [Result]):-
    parseHorizontalLine(RowHLine, Result).

buildBoard([Row|RestElems],
           [RowHLine|RestHLine],
           [RowVLine|RestVLine], BoardLength, Result):-
    buildBoard(RestElems, RestHLine, RestVLine, BoardLength, RestResult),
    parseHorizontalLine(RowHLine, RowHead),
    parseVerticalLine(RowVLine, Row, RowContent),

    Result = [RowHead, RowContent|RestResult].

horizontalLines(0, _):- !,fail.
horizontalLines(1, [[h_line],[h_line]]):-!.
horizontalLines(N, LinesBoard):-
    N2 is N-2,
    horizontalLines(N2,LinesBoardRest),
    setListAt(Top, h_line, N),
    horizontalLinesAux(N, LinesBoardRest, Center),
    setListAt(Bottom, h_line, N),
    append([Top], Center, TC),
    append(TC, [Bottom], LinesBoard).
```

```
horizontalLinesAux(_,[],[]).
horizontalLinesAux(N, [Row|Rest], Center):-
    N1 is N-1,
    length([Row|Rest],N1),
    !,
    horizontalLinesAux(N, Rest, CenterRest),
    append([h_line],Row,HLineRest),
    append(HLineRest,[h_empty],NewRow),

    append([NewRow], CenterRest, Center).

horizontalLinesAux(N, [Row|Rest], Center):-
    !,
    horizontalLinesAux(N, Rest, CenterRest),

    append([h_empty],Row,HEmptyRest),
    append(HEmptyRest,[h_empty],NewRow),

    append([NewRow], CenterRest, Center).


verticalLines(0, _):- !,fail.
verticalLines(1, [[v_empty,v_line]]):-!.
verticalLines(N, LinesBoard):-
    N2 is N-2,
    verticalLines(N2, LinesBoardRest),
    setListAt(ATop, v_empty, N),
    append(ATop, [v_line], Top),
    verticalLinesAux(N, LinesBoardRest, Center),
    N1 is N-1,
    setListAt(CBottom,v_empty,N1),
    append([v_line], CBottom, LBottom),
    append(LBottom, [v_line], Bottom),
    append([Top], Center, TC),
    append(TC, [Bottom], LinesBoard).
    verticalLinesAux(_,[],[]).
    verticalLinesAux(N, [Row|Rest], Center):-
        verticalLinesAux(N, Rest, CenterRest),

        append([v_line],Row,VLineRest),
        append(VLineRest,[v_line],NewRow),

        append([NewRow], CenterRest, Center).

    parseHorizontalLine([], [' ']).
    parseHorizontalLine([R1|Rest], Result):-
        parseHorizontalLine(Rest,Result1),
        meaning(R1, Ascii),
        Result = [Ascii|Result1].

    parseVerticalLine([R1], [], Result):-
        meaning(R1, Ascii),
        Result = [Ascii].
    parseVerticalLine([R1|Rest], [Elem|Tail], Result):-
        parseVerticalLine(Rest, Tail, Result1),
        meaning(R1, Ascii),
        meaning(Elem, TElem),
        Result = [Ascii,TElem|Result1].


    displayBoard([]).
    displayBoard([Row|Tail]):-
        write('               '),
        displayRow(Row), nl,
        displayBoard(Tail).

    displayRow([]).
    displayRow([Elem|Rest]):-
        write(Elem),
        displayRow(Rest).
```

misc.pl

```
splitByLength(Limit,List,SplitList):-
    splitByLength(Limit,List,SplitList,0).

splitByLength(Count, _, [],Count).
splitByLength(Limit, [Value|Rest], SplitList,Count) :-
    Count1 is Count+1,
    splitByLength(Limit, Rest, SplitList_rest,Count1),

    append([Value], SplitList_rest, SplitList).

getKeyIndexedList(0, []):-!.
getKeyIndexedList(N, List):-
    Next is N-1,
    getKeyIndexedList(Next,Rest),
    append(Rest,[N],List).

getColElems(_,_,[],[]):-!.
getColElems(NColumn, BoardLength, List, Result):-
    splitByLength(BoardLength, List, SplitElems),
    append(SplitElems,Rest,List),

    getColElems(NColumn, BoardLength, Rest, TmpResult),

    nth1(NColumn, SplitElems, Value),
    append([Value], TmpResult, Result).

setListAt([], _, 0):-!.
setListAt([Value|Tail], Value, N):-
    N1 is N-1,
    setListAt(Tail, Value, N1).

placeKeys([], [], _).
placeKeys([Elem|Rest], Result, Key):-
    placeKeys(Rest, ResultTail, Key),
    if_else(nth1(Elem, Key, Atom),
        Result = [Atom|ResultTail],
        Result = [empty|ResultTail]).

setListAt([], _, 0):-!.
setListAt([Value|Tail], Value, N):-
    N1 is N-1,
    setListAt(Tail, Value, N1).

placeKeys([], [], _).
placeKeys([Elem|Rest], Result, Key):-
    placeKeys(Rest, ResultTail, Key),
    if_else(nth1(Elem, Key, Atom),
        Result = [Atom|ResultTail],
        Result = [empty|ResultTail]).

listToMatrix([], _, []).
listToMatrix(List, Size, [Row|Matrix]):-
  listToMatrixRow(List, Size, Row, Tail),
  listToMatrix(Tail, Size, Matrix).

listToMatrixRow(Tail, 0, [], Tail).
listToMatrixRow([Item|List], Size, [Item|Row], Tail):-
  NSize is Size-1,
  listToMatrixRow(List, NSize, Row, Tail).


if_else(Condition, If, _Else) :- Condition, !, If.
if_else(_, _, Else) :- Else.

not(X) :- X, !, fail.
not(_X).

switch(X, [Case:Then|Cases]) :-
    ( X=Case ->
        call(Then)
    ;
        switch(X, Cases)
    ).
```