Artificial Intelligence course
University of Parma - A.Y. 2020/2021

Skyscrapers in Prolog

Author: Francesco Vetere
e-mail: francesco.vetere@studenti.unipr.it

Contents

1 Logic Programming

Logic programming is a **declarative** programming paradigm: being declarative means that, generally speaking, the programmer provides the properties that the desired solution should have, rather than specifying the actual sequence of operations needed in order to obtain that solution (which is typical of the imperative paradigm).

In particular, logic programming is based on **formal logic**: basically, every logical program is composed by a list of **facts** and **rules**.
Given a certain goal, the idea is to demonstrate the truth of the goal using the knowledge base formed by facts and rules.

A key concept in logic programming is the one of **unification**.

In logic, unification is the algorithmic procedure used in solving equations involving symbolic expressions.
In other words, by replacing certain sub-expression variables with other expressions, unification tries to identify two symbolic expressions.

We say that two terms unify if they are the same term or if they contain variables that can be uniformly instantiated with terms in such a way that the resulting terms are equal.

In Prolog for example, when we try to unify two terms, the interpreter performs all the necessary variable instantiations, so that the terms really are equal afterwards: if the unification succeeds, Prolog also gives us the value of the instantiated variables.

2 Prolog

2.1 Introduction to Prolog

Prolog is a logic programming language associated with artificial intelligence and computational linguistics, that has its roots in **first-order logic**, a formal logic, and uses a declarative style.

The language was implemented in the 70s by Alain Colmerauer, and since then it's been widely used for theorem proving, expert systems, automated planning and so on.

A Prolog program is generally composed by:

- **Facts** (absolute truths about the domain)

  ```
  father(a, b).
  ```

- **Rules**

  ```
  grandfather(X, Y) :- father(X, Z), father(Z, Y).
  ```

Once the program has been written, we can ask the Prolog interpreter to solve a particular query.

In order to satisfy the goal, a **backward chaining** mechanism it's used:
If a fact that matches the query is known, the goal is satisfied;
Otherwise, for each rule whose consequences meet the question, the interpeter tries to test if each rule premise satisfies the query.

2.2 Basic Prolog example

Let's see a basic example of a Prolog program.

```
%%%%%%%%%%%%%
%%% Facts %%%
%%%%%%%%%%%%%

% Fact 1
father(a, b).

% Fact 2
father(b, c).


%%%%%%%%%%%%%
%%% Rules %%%
%%%%%%%%%%%%%

% Rule 1
% brother(X, Y) ==> true if X and Y have the same father, and X is not Y

brother(X, Y) :- father(Z, X), father(Z, Y), X \= Y.

% Rule 2
% grandfather(X, Y) ==> true if X is father of a generic Z,
%                       and that Z is father of Y

grandfather(X, Y) :- father(X, Z), father(Z, Y).
```

Once the program has been loaded, we can ask for some queries to be solved.

```
?- grandfather (a, c).
true.

?- grandfather (X, c).
X = a.
```

3 Skyscrapers puzzle

3.1 Rules

The goal of this project is to implement the classic **Skyscrapers puzzle** using Prolog.

Skyscrapers is a game in which a matrix is given, and the goal is to tell if that matrix is compliant or not according to some **rules**:

- The numbers inside the matrix represent the heights of buildings.
- The numbers along the sides represent how many skyscrapers a person standing in that spot can see.
- Each row or column contains each number only once.

Here are showed a few examples of how the clues help us to see which skyscrapers we should be able to see:

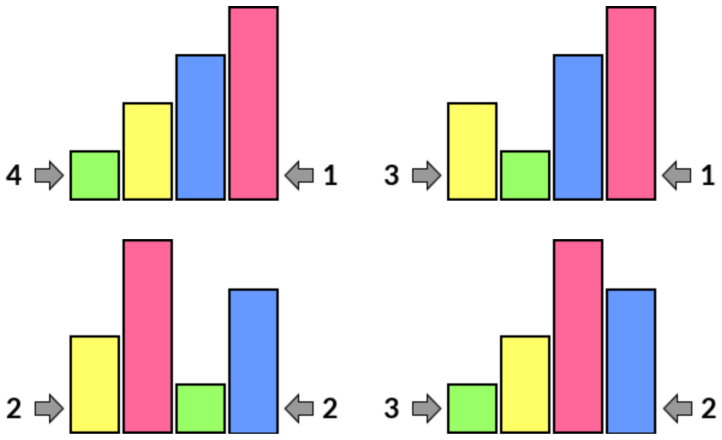

Figure 1: Skyscrapers example

3.2 Example of compliant matrix

So, for example, this matrix is compliant with the rules of the game:

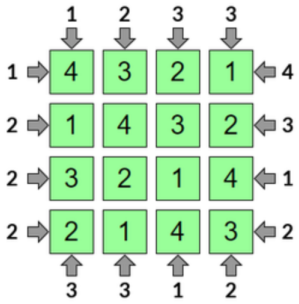

Figure 2: Skyscrapers compliant example

3.3 Matrix format

In order to represent a matrix in a `.txt` file, a particular format has been chosen:

```
<N>
<M[0,0]> ... <M[0,N-1]>
...
<M[N-1,0]> ... <M[N-1,N-1]>
```

So, for example, the following file `3x3.txt` represents a compliant matrix:

```
5
0 0 0 0 0
0 1 2 3 1
0 3 1 2 0
2 2 3 1 0
0 0 0 3 0
```

(Note: 0's on the border are ignored)

4 Implementation

4.1 Program

In order to implement the puzzle in Prolog, a program `skyscrapers.pl` has been written.
It contains some basic predicates on lists and matrices, as well as some other predicates specific for the problem.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%% Predicati di utilita' su liste %%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% member(elem, L) ==> true se L contiene elem
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).

% append(L, R, RIS) ==> RIS  la lista L concatenata con R
append([], R, R).
append([H | T], R, [H | TEMP]) :- append(T, R, TEMP).

% reverse(A, B) ==> B  la lista A rovesciata
reverse([], []).
reverse([H | T], RIS) :- reverse(T, TEMP), append(TEMP, [H], RIS).

% length(A, LEN) ==> LEN  pari alla lunghezza della lista A
len([], 0).
len([_ | T], RIS) :- len(T, TEMP), RIS is TEMP + 1.

% removeFirstLast(A, RIS) ==> RIS  la lista A privata del primo e dell'ultimo
    elemento
removeFirstLast(L, RIS) :- removeFirst(L, TEMP), removeLast(TEMP, RIS).

% removeFirst(A, RIS) ==> RIS  la lista A privata del primo elemento
removeFirst([], []).
removeFirst([_], []).
removeFirst([_ | T], T).

% removeLast(A, RIS) ==> RIS  la lista A privata dell'ultimo elemento
removeLast([], []).
removeLast([_], []).
removeLast(X, Y) :- reverse(X, [_ | T]), reverse(T, Y).

% unique(L) ==> true se L contiene valori unici
unique([]).
unique([H | T]) :- \+ member(H, T), unique(T).

% uniqueFirstLast(L) ==> true se L, privata del primo e ultimo elemento (quelli
    di bordo!) contiene valori unici
uniqueFirstLast(L) :- removeFirstLast(L, LNew), unique(LNew).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%% Predicati di utilita' su matrici %%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% uniqueMatrixFirstLast(M) ==> true se la matrice M contiene liste uniche,
     private del primo e ultimo elemento (quelli di bordo!)
uniqueMatrixFirstLast([]).
uniqueMatrixFirstLast([H | T]) :-
    uniqueFirstLast(H), uniqueMatrixFirstLast(T).

% isMatrixSquare(M) ==> true se M  almeno 2x2 ed  quadrata
isMatrixSquare([H | T]) :- len(H, COLS), len([H | T], ROWS),
                          ROWS >= 2, COLS >= 2, ROWS == COLS.

% transposeMatrix(M, MT) ==> MT contiene la trasposta di M
transposeMatrix([[] | _], []).
transposeMatrix(Matrix, [FirstColumn | TransposedRestMatrix]) :-
    splitMatrix(Matrix, FirstColumn, RestMatrix), % FirstColumn diventa la prima
          riga nella matrice trasposta
    transposeMatrix(RestMatrix, TransposedRestMatrix).

% splitMatrix(Matrix, FirstColumn, RestOfMatrix) ==> Matrix viene divisa in
     FirstColumn e RestOfMatrix
splitMatrix([], [], []).
splitMatrix(
    [[FirstEl | RestFirstRow] | OtherRows],                          % Matrix
    [FirstEl | RestFirstCol],
            % First column
    [RestFirstRow | RestOtherRows]                                              %
          Rest of the matrix
) :- splitMatrix(OtherRows, RestFirstCol, RestOtherRows).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%% Predicati specifici di skyscrapers %%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% countChangesMax(L, RIS) ==> RIS  pari al numero di volte in cui il massimo
     cambia nella lista L (da sinistra a destra)
countChangesMax([0 | T], RIS) :- countChangesMaxAux(T, 0, 1, RIS).
countChangesMax(L, RIS) :- countChangesMaxAux(L, 0, 0, RIS).

% countChangesMaxAux(L, RIS) ==> RIS  pari al numero di volte in cui il massimo
     cambia nella lista L (da sinistra a destra)
%                                                                  con massimo corrente
     pari a CurrentMax e accumulatore Acc
countChangesMaxAux([H | T], CurrentMax, Acc, RIS) :-
    H > CurrentMax,
    AccNew is Acc + 1,
    countChangesMaxAux(T, H, AccNew, RIS).
```

```
countChangesMaxAux([H | T], CurrentMax, Acc, RIS) :-
    H =< CurrentMax,
    countChangesMaxAux(T, CurrentMax, Acc, RIS).

countChangesMaxAux([], _, Acc, Acc).

% isListCompliant(L) ==> true se la lista L rispetta il vincolo del gioco
     skyscrapers previsto per ogni riga/colonna:
%                                       il primo elemento della lista deve
     essere uguale al numero di massimi incontrati
%                                       lungo il resto della lista
isListCompliant([0 | _]).
isListCompliant([H | T]) :- countChangesMax(T, MAXS), H == MAXS.

% checkRuleForward(M) ==> true se la matrice M rispetta il vincolo del gioco
     skyscrapers per ogni riga, da sx a dx
checkRuleForward([]).
checkRuleForward([H | T]) :-
    isListCompliant(H), checkRuleForward(T).

% checkRuleBackward(M) ==> true se la matrice M rispetta il vincolo del gioco
     skyscrapers per ogni riga, da dx a sx
checkRuleBackward([]).
checkRuleBackward([H | T]) :-
    reverse(H, HReverse), isListCompliant(HReverse), checkRuleBackward(T).

% isMatrixCompliant(L) ==> true se la matrice M rispetta le regole del gioco
     skyscrapers
isMatrixCompliant(M) :-
    isMatrixSquare(M),
                      % controllo che la matrice sia quadrata

    removeFirstLast(M, M1), uniqueMatrixFirstLast(M1),  % controllo che all'
         interno dei bordi vi siano valori unici:
                                                        % elimino prima e
                                                             ultima riga, e
                                                             controllo se le
                                                             restanti righe
                                                        % contengono valori
                                                             unici a meno di
                                                             prima e ultima
                                                             colonna

    checkRuleForward(M),
                  % controllo la regola da sx a dx
    checkRuleBackward(M),
                  % controllo la regola da dx a sx

    transposeMatrix(M, MT),

    checkRuleForward(MT),
                  % traspongo e controllo la regola da sx a dx
    checkRuleBackward(MT).
                  % traspongo e controllo la regola da dx a sx
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%% Lettura della matrice da file %%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% read_list_str(L, F) ==> L  la lista letta dal file F
read_list_str(L, F):- see(F), read_list_str(L), seen.
read_list_str([S | R]):- read_atom(S), S\==end_of_file, !, read_list_str(R).
read_list_str([]).
read_atom(end_of_file):- at_end_of_stream, !.
read_atom(S):- read_list_char(L), name(S, L).
read_list_char([C | R]):- get0(C), C\==10, C\==32, C\==(-1), !, read_list_char(
    R).
read_list_char([]).

% list_to_matrix(List, N, Matrix) ==> Matrix  la matrice N x N costruita a
    partire da List
list_to_matrix([], _, []).
list_to_matrix(List, Size, [Row | Matrix]):-
   list_to_matrix_row(List, Size, Row, Tail),
   list_to_matrix(Tail, Size, Matrix).

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

% build_matrix(File, Matrix) ==> Matrix  la matrice letta dal file File
build_matrix(File, Matrix) :- read_list_str([H | T], File), list_to_matrix(T, H
    , Matrix).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% Predicato principale %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% skyscrapers(File) ==> true se File contiene una matrice in accordo con le
    regole del gioco
skyscrapers(File) :- build_matrix(File, Matrix), isMatrixCompliant(Matrix).
```

4.2 Tests

Now we can test our program, giving it some compliant matrices in input, and verifying that the main goal is satisfied.

```
> swipl

?- consult('skyscrapers.pl').

?- skyscrapers('matrices/3x3.txt').
   ==> true.

?- skyscrapers('matrices/4x4.txt').
   ==> true.

?- skyscrapers('matrices/5x5.txt').
   ==> true.

?- skyscrapers('matrices/6x6.txt').
   ==> true.
```

References

[1] Gabbrielli M., Martini S.
Linguaggi di programmazione. Principi e paradigmi.
McGraw-Hill, 2011

[2] *SWI Prolog*
`https://www.swi-prolog.org/`

[3] *Tracing simple program in Prolog*
`http://www.ablmcc.edu.hk/~scy/prolog/index_e.htm`

[4] *Skyscrapers puzzle*
`https://www.brainbashers.com/skyscrapershelp.asp`