INF8215 – I.A.: méthodes et algorithmes

TP2
Résolution des problèmes avec Prolog

Cynthia Berenice Castillo Millán
Matricule: 1878153

Date de remise:
25 mars 2017

Table de matières

1. Project 1: Devinette de personnes et devinette d'objets

1.1. Objectives

The objectives of the project can be defined as:

- ❏ Find an object or person (only one) from a list by asking yes/no questions.
- ❏ Easy expansion of the object list in case of needed.
- ❏ Minimize the number of questions needed to find the object / person.
- ❏ Establish if the object is part of a certain class when asked for it..
- ❏ Establish if an object or person is part of the list when asked for it.

1.2. Implementation

Each *devinette* has its own prolog program but they work in the same way. Each program can be divided in three main parts:

1. Database (list): the objects and their "attributes" are defined as facts. Some of these attributes are not defined because they are constants (for example, if a person is male or female).

```
musicien(john_lewis, homme, jazz).
musicien(wolfgang_amadeus_mozart, homme, classique).

musique(jazz).
musique(classique).
```

2. Questions: there are two types of questions. The first one only displays information to the user and work as a *break* in case the answer of the user is negative. The second save a value in case of a positive answer.

```
ask(musicien):-
    write('Est un musicien? '),
    read(Reponse),
    Reponse = 'oui'.

ask(musicien, Z):-
    format('Le type de musique est ~w ? ',[Z]),
    read(Reponse),
    Reponse = 'oui'.
```

3. Search: Once defined all the objects and questions, they are ensembled them with the original questions *objet(X)* or *personne(X)* which search among the questions until they find the object or return false.

1.3. Solution

The first problem is to identify only one of the elements of the list. To solve that problem some *attributes* were added. The more objects there are, the more specific the search is needed to be. In this case, only three attributes were defined for each object or person.

The number of questions needed to find the answer is related to how big is the *class* the object or person belongs to. Also, the number of questions asked is related to the order in which the are asked.

In order to improve the results (minimize the questions needed) , two questions (in the case of *devinette personnes,* just one in the case of objects) are asked to:

- ❏ Establish if the person searched is male or female (*devinette personnes*).
- ❏ Establish if it a real person or a character (*devinette personnes*).
- ❏ Establish if it's an electric object or not (*devinette objets*).

The intention is to make a search that resembles a binary search (at first, at least). This pre-selection before starting with more specific questions helps to reduce the options as most as possible from the very beginning.

After that, the questions asked are ordered from the biggest groups to the smallest. Taking advantage of the probability, is more possible that the object search is found in one of the most numerous groups than in a group of only one element.

1.4. Results analysis

Given the current values of the data base of the program:

Personnes

Solution improvement	**# of questions Best case scenario**	**# of questions Worst case scenario**	**Average # of questions**
No improvement (brute force)	1	22	11
Female / Male	2	15	8.5
real / character	2	17	9.5
Female / Male +	3	12	7.5

real / character			

Objects

Solution improvement	**# of questions <u>Best case scenario</u>**	**# of questions Worst case scenario**	**Average # of questions**
No improvement (brute force)	1	22	11
Electric or not	2	13	7.5

In addition to those bigger subdivision mades to the database information, several more specific subdivision were made. In this small example, the difference is not as noticeable but the bigger the database gets, the more important it becomes to make more and subdivisions of the information to make the search resemble as much as possible to a binary search in order to increase its efficiency.

2. Projet 2: Nonograms

2.1. Implementation

In order to solve the problem, the first thing needed is to get the size of the table, to do so, we get the number of list of any of the list of constraints given which determine the number of spaces per line or column (the table size). The function is similar for both, valid_lines and valid_columns:

```
valid_lines(LineSpecs, X):-
        length(LineSpecs,Size), %get lines length
        valid_lines(LineSpecs, Size, X).
```

To start filling the table, the program do two main things:

1. Creates a lists of lists with all the possible valid lines given the constraints
2. Select a list of lines that fulfill the column constraints.

If both, line constraints and column constraints are fulfilled, then it's a valid solution for the nonogram.

Valid_lines gets all the possible valid lines for each constraint. To do so, it calls a function which return a list with all the possible valid lines for a given constraint:

```
%Creates all the possible valid lines that fulfill the constrain
ts
%Constraint = list with constraints for a line
%size of the line
%Lines =  list of lists with possible valid lines for constraint
line([], _, _).
line(Constraint, Size, Lines):-
        length(Lines,Size), %make an empty line of variables
        Lines ins 0..1,  %constraint the value of the variables
        label(Lines), %ground the variables with all possible
        %combinations that fulfill the constraints
        valid_seq(Constraint, Lines). %verifying if the line is
 valid
```

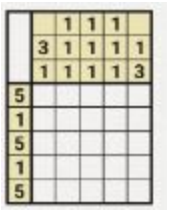
After getting all the valid lines for a certain list of constraints, it saves its in a matrix (list of lists) and continues the search for the next list of constraints.

```
%Creates all the valid lines with the given constraints
%[Head|Tail] = Constraints
%Size =  size of each line
%[X|L] = list of lists with all possible valid lines
valid_lines([],_, []).
valid_lines([Head|Tail], Size, [X|L]):-
        line(Head, Size, X),%Creates a valid line
        valid_lines(Tail,Size, L).
valid_lines(LineSpecs, X):-
        length(LineSpecs,Size), %get lines length
        valid_lines(LineSpecs, Size, X).
```

Once all the possible lines were made, valid columns search in all the lists one that fulfill the constraints for each column. To do so, a counter for the columns is implemented each time a correct column is found, so it continues checking the following columns of the list.

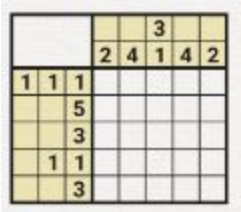
```
valid_columns([],_, _, _).
valid_columns([Head|Tail], Size, Counter, Lists):-
        Counter =< Size,
        extract(Counter, Lists, X), %get a column
        valid_seq(Head, X), %verifying its a valid column
        C #= Counter + 1, %if it is, find the next column
        valid_columns(Tail, Size, C, Lists).
```

2.2. Solutions



```
?- logicPrb([[3,1], [1,1,1], [1,1,1], [1,1,1], [1,3]], [[5], [1], [5], [1],
 [5]], X).

Found nonogram:
# # # # #
# . . . .
# # # # #
. . . . #
# # # # #
X = [[1, 1, 1, 1, 1], [1, 0, 0, 0, 0], [1, 1, 1, 1, 1], [0, 0, 0, 0, 1], [1
, 1, 1, 1|...]]
```



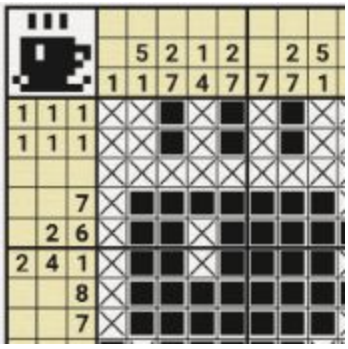
```
?- logicPrb([[2], [4], [3,1], [4], [2]], [[1,1,1], [5], [3], [1,1], [3]], X
).

Found nonogram:
# . # . #
# # # # #
. # # # .
. # . # .
. # # # .
X = [[1, 0, 1, 0, 1], [1, 1, 1, 1, 1], [0, 1, 1, 1, 0], [0, 1, 0, 1, 0], [0
, 1, 1, 1|...]]
```

2.3. Solution analysis

The time complexity to find the results is O(n!) order. The bigger the table gets, the more time needed to find a solution because it needs to explore all the possible permutations for a given constraint in each line. One more line in the table increase considerably the time of processing.

For a 5 x5 table, the time to find a result was about half a second, while for a 8 x 8 table, took around two minutes. This is the result for a part of the coffee cup:

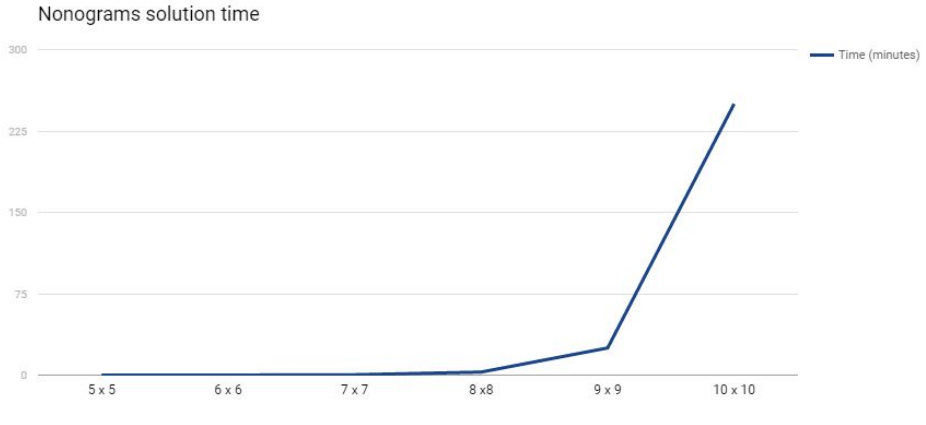


```
?- logicPrb([[0], [5], [2,5], [1,2],[2,5], [5], [2,5],[5]], [[1,1,1],[1,1,1
],[0], [7], [2,4],[2,4], [7], [7]],X).

Found nonogram:
. . # . # . # .
. . # . # . # .
. . . . . . . .
. # # # # # # #
. # # . # # # #
. # # . # # # #
. # # # # # # #
. # # # # # # #
X = [[0, 0, 1, 0, 1, 0, 1, 0], [0, 0, 1, 0, 1, 0, 1|...], [0, 0, 0, 0, 0, 0
|...], [0, 1, 1, 1, 1|...], [0, 1, 1, 0|...], [0, 1, 1|...], [0, 1|...], [0
|...]] .
```

Taking this values in consideration, we could expect that the time needed to find the complete result of the coffee cup (10 x 10) will need to explore 10! possible solutions, which would take around 4 hours.

Size table table	Time (minutes)
5 x 5	0.008
6 x 6	0.05
7 x 7	0.35
8 x8	2.8
9 x 9	25
10 x 10	250



And we can expect that the time needed to solve the last example, the 15 x 15 table, would take over 25 years to solve!