INF8215 – Intelligence artificielle : méthodes et algorithmes

Lab 2: Prolog, logic programming, constraint programming

Introduction

This lab is divided into two independent parts: a first one about constraint programming and a second one about logic programming. We will first make use of *Prolog*, which is a logic functional programming language. In this part, we will be dealing with knowledge bases containing predicates and rules. Prolog enables us to make use of these tools to extract results and information.

In a second part, we will be dealing with constraint programming (CP). This programming paradigm enables us to solve combinatorial problems efficiently using a certain set of tools. To do this, we will make use of *MiniZinc* along with the Gecode solver.

Set up

Both *Prolog* and *MiniZinc* are available to use on your workstations. To launch them, use the terminal and run either one of these commands:

- `swipl`
- `MiniZincIDE`

To install *Prolog* on your own personal machines, visit the official SWI-Prolog website. To install *MiniZinc*, visit the official GitHub repository of MiniZinc.

Tip for Windows and macOS:
Add…

- `%PROGRAMFILES%\swipl\bin` (Windows)
- `/Applications/SWI-Prolog.app/Contents/MacOS` (macOS)

… to your `PATH` to have an easier access to Prolog inside *Powershell* or *Command Line* on Windows or *Terminal* on macOS.

Help writing Prolog programs:
Install *Sublime Text 3* on your computers, they can recognize the Prolog syntax. To set this up, press Ctrl+Shift+P (or Cmd+Shift+P on macOS) and run these commands in this exact order :

- `Install Package Control`
- `Package Control: Install Package`
- `Prolog`

To enable the Prolog syntax, go to *View* > *Syntax* > *Prolog* > *SWI-Prolog*.

Important basics

Using MiniZinc

Refer to the examples on Moodle or to the MiniZinc Tutorial for help using *MiniZinc* and understanding its syntax.

Prolog concepts

Prolog is a logical functional programming language. The code does not necessarily run as a sequence of instructions. On the contrary, it runs as a sequence of function calls. In the case of Prolog, we manipulate predicates that are in a *knowledge base*.

Everything in a knowledge base is assumed to be **true**. On the other hand, anything that is not there is automatically **false**.

A knowledge base is composed of *predicates*. These predicates are defined by a list of literals, where each literal takes the form of a `symbol_predicat(constant)`.

Consider the following knowledge base:

```
human(scott).
human(ramona).
human(envy).
human(wallace).
girl(ramona).
enemies(scott, envy).
```

Open a text file and write those lines in it. Save the file as `example.pl`.

In a terminal window, go to directory containing your file and execute:

```
swipl -s example.pl
```

Alternatively, you can first run `swipl` on the terminal, without loading the file. You can do so afterwards by using one of the following commands (this is also useful for reloading a modified file):

```
?- [example].
?- consult(example).
```

Our knowledge base contains only one predicate: `human`. This predicate is defined on 3 literals. Elements in parentheses are called “atoms” (constants). Instructions of any kind in Prolog can be written using multiple lines, but they all must end with a dot (`.`).

Querying a knowledge base on Prolog:
Once loaded, it is possible to query our knowledge base by writing literals following the question mark “`?-`”.

Now, try querying the knowledge base with `human(scott)` and `human(ramona)`. You should receive the answer `true`. Since these facts exist in our knowledge base, they are `true`.

Following that, query the database with `human(envy)`. You should expect the answer `false`. Indeed, this literal does not exist, and `envy`, within this framework, is not a `human`. Now try to execute `singer(envy)`. You should be getting an error at this point informing you that `singer/1` is undefined. In fact, the predicate `singer` was never defined, which is different from a *false* answer. While `false` is the result of a query that may be valid, this error is an unexpected behavior and is, therefore, a programming error to avoid.

Information: predicates and rules are designated by Prolog as `symbol/n`, where `n` is the number of parameters `symbol` takes. `symbol` can be defined to be compatible with different numbers of parameters, and the corresponding prototype is designated by the appropriate `n`.

Variables:
It is possible to use variables to enhance queries for more information. To define a variable, use a capital letter at the beginning of its name.

Suppose we want to know who the humans in our knowledge base are. It is sufficient to execute one of the following queries:

```
?- human(X).
?- human(ThisIsAVar).
```

The answer of Prolog should be (if we use `X`):

```
X = scott ;
X = ramona ;
X = envy ;
X = wallace.
```

(Press `;` after each line to display the next one, use `.` to stop the search)

Thus, prolog will look for all atoms that can be attributed to `X`, and print them one by one.

Boolean operators:
It is possible to write queries as conjunctions or disjunctions of literals. To this end, we need the *AND* and *OR* operators. In Prolog, *AND* is a comma (`,`) while *OR* is a semi-colon (`;`).

Using this, we can search for all humans who are also girls by calling the following query:

```
?- human(X), girl(X).
```

We'd get:

```
X = ramona ;
false.
```

(`false` here indicates the end of the search)

Note that, even thought `envy` is a girl in real life, since we did not indicate it in the knowledge base, this information is not present in the output.

Moreover, our knowledge base was written in such a way that the call to `girl/1` only would have been enough to have all the girls without the need for `human/1`.

Rules and procedures:
Using what we have learned so far on how Prolog works, we can define rules to perform actions or obtain information.

A rule is generally defined as follows:

```
symboleRegle(parametres) :- litteraux.
```

Consider the literal `enemies(scott, envy).` which says Scott and Envy are enemies. If we query our knowledge base as shown below, the answer will be *false* since the predicate `enemies/2` was never defined in that shape:

```
?- ennemis(envy, scott).
```

However, this is unexpected because this contrary to the idea that Scott and Envy are, in fact, enemies. To fix this, we can supplement `ennemis/2` with a rule that will bypass this:

```
enemies(X, Y) :- enemies(Y, X).
```

Using this rule, when Prolog does not find the literal with the indicated atom order in the knowledge base, it will backtrack to the rule above which reverses their order. Prolog then picks up its search with the inverted parameters and finds the adequate literal.

Of course, it is possible to write a conjunction or disjunction of literals in the definition of rules to make them more efficient. For example, if we are only looking for Scott's human enemies, we will be using the following rule:

```
humanEnemiesScott(X) :- (ennemis(X, scott) ; ennemis(scott, X) ), humain(X).
```

Anonymous variables:
Sometimes, in some rules, there is a need to store an element in a temporary variable that is not going to be used. Instead of using a normal variable (which would trigger a singleton variable warning), we use anonymous variables. Anonymous variables are an underscore (_). Each of these anonymous variables is independent.

Debugging:
Prolog offers a debugging tool which gives an inside look into the Prolog search procedure at each step (trace). This makes it possible to detect errors more efficiently. In order to enable this functionality, run the following command:

```
trace.
```

To disable this mode, run:

```
notrace, nodebug.
```

Comments:
To include comments in the *pl* files, add the % symbol at the beginning of each line.

Tests and unification

Tests involving two variables:
We can compare variables as we would normally do in any other programming language. The comparison may result in a `true` or `false`, and the syntax is as follows:

- X==Y : `true` if X and Y are equal (same *atom*).
- X\==Y : `true` is X et Y are **not** equal.
- X<Y or X=<Y : evaluates X and Y and returns `true` if X is smaller than (resp. smaller or equal to) Y.
- X>Y or X>=Y : evaluates X and Y and returns `true` if X is larger than (resp. larger or equal to) Y.

Unification:
Unification matches an expression to a similar one. For example, X=Y is an expression that can be unified to 2=3 because the two expressions are *similar*. After unification, X take on 2 and Y will take on 3.

Unification can only take place if the elements can be changed to make the correspondence. Thus, 3+3 and 2+4 cannot be unified even though the expressions are similar.

Unification allows to perform a *Pattern Matching*, eg:

- X+Y unified to m(p)*6+5/9 matches m(p)*6 to X and 5/9 to Y.
- X+Y unified to 6+7+8+9 matches 6 to X and 7+8+9 to Y.
- Y*4 unified to 0*X matches 4 to X and 0 to Y.

Let M and N be expressions:

- M=N: true if M and N are unifiable and, if so, performs unification.
- M\=N: true if M and N are **not** unifiable. If they are, unification is not accomplished.
- M is N: evaluates N and unifies the result to M (ie, stores the result in M).

Recursion and lists

Lists:
Prolog allows to handling lists of elements. Their general syntax is as follows:

```
[item1, item2, item3, …, itemM, itemP, …, itemN]
```

To work with lists, it is important to know how to divide them. Each list can be unified to an element with the following structure:

```
[Var1, Var2, …, VarM | B]
```

If we unify the two lists at the top, we will have the matching below:

- Var1 takes item1, Var2 takes item2, …, VarM takes itemM.
- B is assigned the remainder of the list, ie [itemP, …, itemN].

Using this technique, along side a recursive formulation, it is possible to manipulate lists in any way we want. It is not possible to index the list or iterate through it.

Recursion:
A rule can invoke itself in its execution. Since Prolog does not accept iterative structures, recursion is the only way to perform iterative operations.

Recursive rules often need at least two definitions: a recursive formulation and a base state. Recursion stops when this base state is reached.

Consider the following example where we want to look for the atom 3 in the list [0, 6, 2, 3, 9, 4] and put it in a variable:

```
lookFor3([3|_], 3).
lookFor3([L|B], R):- lookFor3(B, R).
```

The second definition will, each time, remove an item from the list and call itself again with the remaining elements in the list B. At each subsequent call, Prolog consults the knowledge base, from top to bottom. When lookFor3 is called on a list that starts with 3, the first literal is going to be true. This marks the end of the branch in the search tree and assigns the second parameter (when this one is a variable) the atom 3.

Note that the order here is **important**: with the reverse order, lookFor3 will empty the list, get to a dead end and backtrack in search of another literal that can be satisfied. In this case, the result remains the same, but the search is unnecessarily longer (you can try to visualize this on your machines with trace). In other cases, this can produce errors or unexpected behavior. The code, as presented at the top, finds an answer as soon as a list with a 3 at the beginning is encountered.

Cut:
Sometimes, in some recursive formulations (or elsewhere), it is desired to stop the search as soon as a branch of the search tree is fully explored. In other words, we do not want there to be backtracking in search of another solution. We use the predicate `!/0`, also called *cut*.

Consider the following example of a rule that is *true* if an X is a member of a list:
```
isMember(X, [X|_]).
isMember(X, [Y|B]):- isMember(X, B).
```

Written like this, even though our code works, it is once again unnecessarily inefficient, because the whole list will be tested in search of the element X. To stop the search at the right moment, we rewrite the first rule:
```
isMember(X, [X|_]):- !. % !/0 is a literal like any other, true by def
isMember(X, [Y|B]):- isMember(X, B).
```

Exercise 1: Detective

Consider the following statements:

- The Englishman lives in the red house.
- The Spaniard owns a dog.
- The occupants of the green house drink coffee.
- The Ukrainian drinks tea.
- The green house is located immediately to the right of the white one.
- The sculptor breeds snails.
- The diplomat lives in the yellow house.
- The occupants of the house located in the middle drink milk.
- The Norwegian lives in the leftmost house.
- The physician lives closest to where the fox's owner lives.
- The diplomat lives closest to the house where a horse is kept.
- The violinist drinks orange juice.
- The Japanese person is an acrobat.
- The Norwegian lives closest to the blue house.

Answer these questions…

- Who drinks water?
- Who owns a zebra?

… by considering this problem as a CSP. Create and make use of an adequate model on *MiniZinc* to do this.

Exercise 2: Round-Robin

Consider a tournament where *N* football teams are pitted against one another. During this tournament, each team faces every opponent exactly once. This kind of tournament is called a *Round-Robin tournament*.

All teams simultaneously play in each round. Seeing how every team will play against every other one once, this means there will be exactly *N* – 1 rounds.

Moreover, each team has its own football stadium. A team is said to be *home* if they host their opponent on their own stadium. Otherwise, they are *away*.

A data file, `N14a.dzn`, is given. In it are two variables: `N`, which is the size of the problem instance and `location`, which is an array specifying the locations of all matches. The array is such that `location[i,j] = 1` if `i` faces `j` at home or `location[i,j] = 0` if `i` faces `j` away (ie, `j` play at home).

→ Find an appropriate schedule for this tournament if, additionally, **we won't allow that a team plays 4 successive matches or more either home or away**.

Define an appropriate model for this CSP on *MiniZinc* and solve it.

Find and explain a symmetry in this problem and break it using a redundant constraint. What do you notice as far as solve time goes? Why such a significant difference?

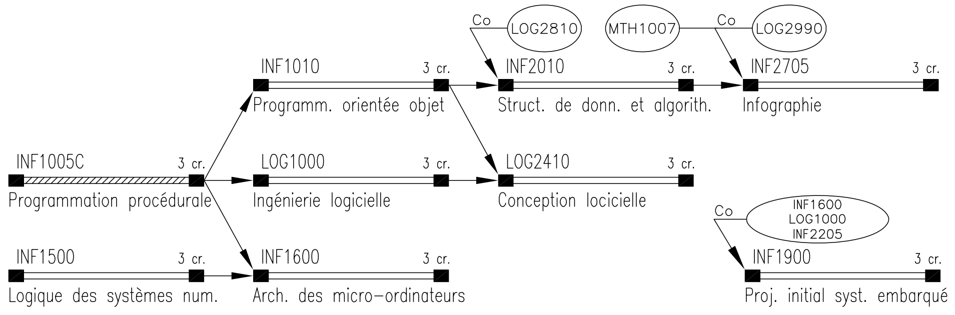
Bonus

Find a global constraint that can be applied in this CSP to account for the home/away rule. Explain how it works and what it does. Why is it better to use it instead? **You are not required to implement this** in your model to get full marks. However, the accuracy and thoroughness of your answer will influence the number of points you get.

Exercise 3: Course requirements

Description

Consider the following coursework:



We'd like to develop a *Prolog* program which, given a course, informs us about what other classes to take prior to be eligible for it.

Example

For the INF1900 course, the answer should be: INF1600, INF2205, LOG1000, INF1005C, INF1500.

Procedure

Define an appropriate and efficient knowledge base for this problem. **It should only contain predicates of at most two parameters**.

Write a rule `completeRequirementsFor/?` which gives a complete set of prerequisites and corequisites to a given course. You are free to implement this however you like. You are also allowed to use *Prolog* factory functions. **You cannot use external libraries**.

At the end, we require a clear and easy to interpret output. **Duplicates and other artefacts are not permitted**.

Exercise 4: Akinator

Description

We'd like to implement a riddle solver using a knowledge base on Prolog. The program runs on the assumption that we have a famous person or household object in mind. It then tries to guess who or what that is by asking a series of yes/no questions.

In reality, this task is accomplished by two separate programs, each one with a distinct knowledge base: one for people and the other for objects.

People:

Michael Jackson
Mikhail Gorbachev
Jennifer Lawrence
Hideo Kojima
Banksy
Lara Croft
Mario
J. K. Rowling
Lady Gaga
Quentin Tarantino
Joseph Staline
Dwight D. Eisenhower
Cleopatra
Victor Hugo
Jesus
Ayrton Senna
Moses
Fernando Alonso
Pope Francis
James Bond
Denzel Washington
Richard Nixon

Objects:

Vacuum
Computer
Phone
Fork
Broom
Cactus
Plate
Oven
Range
Coffee machine
Toaster
Table
Pan
Shampoo
Dishwashing detergent
Bed
Key
Wallet
Backpack
Piano
Lamp
Paper

Important considerations

You must make sure your knowledge base is sufficiently robust and flexible to make adding further elements to it as effortless as possible.

It is also important that your knowledge base be compact and efficient. Your program must be able to make a valid guess asking the least amount of questions.

Example

To help you get started, here's a small example with 4 people:

```
ask(governs, Y) :-
      format('~w governs? ', [Y]),
      read(Reply),
      Reply = 'yes'.
ask(musician, X) :-
      format('~w is a musician? ', [X]),
      read(Reply),
      Reply = 'yes'.
ask(singer, X) :-
      format('~w is a singer? ', [X]),
      read(Reply),
      Reply = 'yes'.

person(X) :- politician(X).
person(X) :- artist(X).

artist(X) :- ask(singer, X), singer(X).
artist(X) :- ask(musician, X), musician(X).

politician(X) :- governs(X, Y), country(Y), ask(governs, Y).

singer(celine_dion).
musician(john_lewis).
governs(stephen_harper,canada).
governs(barack_obama,usa).
country(canada).
country(usa).
```

Execution

Running your program should require consulting your knowledge base using one of the following enquiries:

```
?- person(X).
?- object(X).
```

Your program should also allow direct verification of object class. For example, running the following enquiry…

```
?- appliance(oven).
```

… should return *true* in this case.

Submission instructions

Write a PDF report containing detailed information about your method, your reasoning, your choices, and your implementations. **Do NOT include any code**: the report should rather supplement it. **Be as concise and precise as possible**.

Turn in this report, along with any relevant *pl* and *mzn* files, on Moodle, in an archive.

→ *Deadline: November 11th, 2018 at 11:55pm.*

Scale

- Exercise 1: 2.5 pts
- Exercise 2: 6 pts + 2 pts
- Exercise 3: 6.5 pts
- Exercise 4: 5 pts

** This lab assignment was partly based on the prior work of Gilles Pesant, Pierre Hulot and Rodrigo Randel.*