UvA Matching 2017: Problem Solving with Prolog (BSc KI)

Ulle Endriss

Institute for Logic, Language and Computation
University of Amsterdam

1 Purpose

AI is all about problem solving: developing general methods for getting machines to perform tasks that we tend to associate with "intelligence" when performed by a human being. *Prolog* is an important AI programming language that is particularly useful for the development of such methods. First-year AI students learn about Prolog in the course *Problem Solving and Search*. In this UvA Matching module you will get a first glimpse at Prolog and you will see how it can be used for AI-style problem solving. Our main example for a problem to be solved will be a game known as the *Towers of Hanoi*.

2 Planning & Information

There will be a 2-hour lecture on *Problem Solving with Prolog* on the first day (Friday), followed (after lunch) by a 3-hour computer lab session. At the end of that lab session you will have to submit your solutions to the assignment below. On the final day (Tuesday) there will be 1-hour (plenary) tutorial session in which we will discuss your solutions and answer any further questions that you may have.

All relevant information for this module (precise timetable, room information, teaching materials) is available at the following website:

`https://staff.science.uva.nl/u.endriss/teaching/uva-matching/`

The deadline for submitting your assignment is 17:00 on Friday, 9 June 2017. Your personal *study advice* for the UvA's *BSc Kunstmatige Intelligentie* will (to 50%) be based on the results you achieve for this assignment.

3 Getting Started

If you are using your own laptop, make sure SWI-Prolog is installed. If you are using one of the UvA desktop computers, log into Linux (not Windows). Then create a new directory (also called folder) with the name `matching`, download the file `hanoi.pl` from the website (and save it in `matching`), create a text file of the form `yourname.pl` (also within `matching`),

start up Prolog, consult both of your files in Prolog, and then try out a few examples. Here are the same instructions in a little more detail, for those of you using an UvA desktop machine (you presumably know how to achieve the same kind of effect on your own laptop):

(1) The computers in the labs run both Windows and Linux. We will work with Linux. In case the previous user has been working with Windows, restart your machine and follow the on-screen instructions to *start Linux*. Then *log in*, using the account details you have received from the UvA.

(2) Open two *terminals* (little windows you can type commands into). To open a terminal, go to the `Applications` menu, choose `System Tools`, and then `Terminal`. Let's call one of the two terminals the *left terminal* and the other the *right terminal*. We will use the left terminal to *run* Prolog programs and the right one to *write* Prolog programs.

(3) Create a subdirectory called `matching` in your home directory (to store all files related to UvA Matching). To do this, type the following command (short for "make directory") into either one of your two terminals:

```
mkdir matching
```

Now switch to this new directory in *both* of your terminals, by typing the following command (short for "change directory") into each one of them:

```
cd matching
```

(4) Download the file `hanoi.pl` from the website and save it in your `matching`-directory.

(5) Create a text file (within your `matching`-directory) in which to store your own program. The name of your file should be the same as your surname, using only lowercase letters, and it should have the file extension `pl`. For example, if, like most people, you are called Joost Jansen, then your personal file should be called `jansen.pl`. Use the editor *Emacs* to create and edit your file. You can start *Emacs* by executing the following command in your *right terminal* (because this is about *writing* programs):

```
emacs jansen.pl &
```

Once in *Emacs*, it should be self-explanatory how to do things such as saving your file. Now copy the following text into your personal file (*exactly* as shown) and save it:

```
% Examples copied from the handout:
test(1) :- writeln('It works!'), writeln('Life is beautiful.').
test(2) :- init(8), move(1,2), move(2,3), move(3,1), test(1).
```

(6) Now we want to *run* programs, so we switch to the *left terminal*. Execute the following command to start Prolog:

```
pl
```

You should see that the prompt has changed to `?-`. In the sequel, all commands to be typed into Prolog are preceded by `?-` (but you don't actually type this in). Importantly, all Prolog commands must end in a full stop (if you don't do this it all goes pear-shaped and you'll need expert help to recover). Now tell Prolog to *consult* the file `hanoi.pl`:

```
?- consult(hanoi).
```

If everything went according to plan, you should see some explanations on how to use the Game Console. Read them (including the information on how to display those explanations again at a later point).

(7) Now *consult* your own personal file in Prolog:

```
?- consult(jansen).
```

If everything went well, Prolog should say so (that is, it should say that it compiled your file successfully and it should not display any error or warning messages). Whenever you change and save your file in *Emacs* you need to re-run the command above, to make sure Prolog knows about the latest version of your program.

(8) Finally, try out a few *examples* to see how Prolog reacts. Here are some suggestions:

```
?- test(1).
?- test(2).
?- init(8).
?- move(1,3).
?- init(5), move(1,2), move(1,2).
?- init(5), sillySolve.
?- init(5), recSolve(5,1,3).
?- init(5), recSolve(5,1,2).
?- init(6), recSolve(6,1,3), recSolve(4,3,2), recSolve(2,2,1).
```

Try to predict what will happen before you actually try it out on the computer.

4 Assignment

Solve the exercises below and save your solutions in your personal file. Some exercises are programming exercises and some are theoretical exercises. For the theoretical ones, put your solutions inside a comment environment (`/* ... */`).

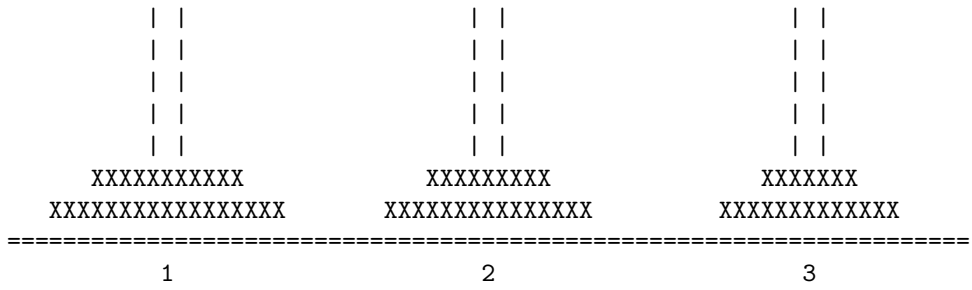
You may not manage to complete absolutely everything during the lab session, and this is not a big problem. However, make sure you check Section 5 (on how to organise your file and on how to submit your solutions) at least one hour before the end of the lab session, to make sure you know exactly what to do when the deadline comes.

Exercise 1. Write a program that has the effect that when the user types in the word "`welcome`" (followed by a full stop), Prolog will write a welcome message on the screen that appropriately reflects your personality. Here is an example (but don't just copy this):

```
?- welcome.
  ******************************************************
  *                                            (\_/)   *
  *  Welcome to the world of Joost Jansen!    (='.'=)  *
  *       Feel free to look around ...        (")_(")  *
  ******************************************************
```

Hint: Have a look at how we have implemented the first `test`-predicate above.

Exercise 2. Find a sequence of Prolog commands such that you end up with the following configuration of the game:



Now write a small program that has the effect that when you type the word `flat` (followed by a full stop) into Prolog, the system will show the sequence of steps starting from the initial configuration (with 6 disks) to the one shown above. To demonstrate that it really works, also copy the (final) output generated by Prolog into your personal file (within `/* ... */`).

Exercise 3. Moving a disk from `A` to `B` only takes an instant in Prolog. To make it possible for users to observe what happens, the Game Console puts Prolog to sleep for 1 second after every move. Thus, the number of seconds it takes Prolog to solve a game is a pretty good approximation of the number of steps required. You can measure the time it takes to execute a command using the predicate `time/1`. Try the following example:

```
?- init(4), time(recSolve(4,1,3)).
```

This creates a game with 4 disks and then solves it (by asking Prolog to use the recursive algorithm to move a tower of size 4 from peg 1 to peg 3). The solving part is getting timed. This should take around 15 seconds: there are $2^4 - 1 = 15$ moves to be executed and each one of them takes around 1 second. As you will see, Prolog not only reports the time taken, but also some other information (which you can ignore at this point).

Use the above method to measure the time it takes Prolog to solve games of size 2, 4, and 6 using the recursive algorithm. Add a small table to your personal file in which you record the runtimes for each of them. Do your experimental findings confirm the theoretical complexity results discussed in the lecture?

Now use the same method to measure the time it takes Prolog to solve games of size 2, 4, and 6 using the iterative algorithm. Recall that the Prolog command for executing the iterative algorithm is `itSolve` (and that, unlike `recSolve`, it does not take any arguments). Copy an example into your personal file showing what you did to measure these times. Then, again, prepare a small table with the runtimes for these problems. Now compare the results for the two algorithms. What can you say about their relative performance?

Exercise 4. Legend has it that somewhere in the jungle near Hanoi, Vietnam, there is an old temple with a real-life version of the *Towers of Hanoi* with 42 disks, made of pure gold. The local monks move one disk each and every day (following an optimal strategy that minimises the number of moves required). They started doing so 1000 years ago. Legend also has it that the world will end once the tower has been fully moved to its target location. How many years do we have left until the end of the world? Explain your answer.

Exercise 5. Let us call a configuration of disks *legal* if no disk is placed on top of a smaller disk (and if all disks are on one of the three pegs). How many legal configurations are there for a game with 2 disks? How many are there for a game with n disks?

Exercise 6. Suppose we don't want to allow moves between the leftmost and the rightmost peg (i.e., between pegs 1 and 3), but only between *direct neighbours*. Can we still solve the game under this restriction? The answer is *yes*. For example, for the case of a tower with just 2 disks, we can do as follows (try it by copy-pasting the text below into Prolog):

```
?- init(2),
   move(1,2), move(2,3), move(1,2),
   move(3,2), move(2,1), move(2,3),
   move(1,2), move(2,3).
```

Try to find a general solution to this problem (and describe your approach, even if you don't manage to implement it). Implement your solution as a predicate called `dirSolve/3` that, just like `recSolve/3`, takes three arguments (the number of disks, the starting position, and the final position). You may want to use our existing implementation of `recSolve/3` (see slides) as a starting point. Once you're done, examples such as the next one (moving a tower of size 4 from peg 1 to peg 3 using only moves between direct neighbours, i.e., only moves involving peg 2) should work:

```
?- init(4), dirSolve(4,1,3).
```

By the way, if you've done everything right, then your algorithm will (rather amazingly!) visit *every possible legal configuration* along the way. Can you prove that this is indeed the case? (*Hint:* Try to analyse the complexity of your algorithm first.)

5 Submission

Make sure that consulting your file produces *no error messages* (ask for help if needed). Put a comment at the top of your file with your name and your email address. Clearly indicate the start of each new exercise with a comment. So your file will look something like this:

```
% UvA Matching 2017: Problem Solving with Prolog (BSc KI)
% Solutions submitted by Joost Jansen, jj@postbakje.nl

% Examples copied from the handout:
test(1) :- writeln('It works!'), writeln('Life is beautiful.').
test(2) :- init(8), move(1,2), move(2,3), move(3,1), test(1).

% Exercise 1
...

% Exercise 6
/* To come up with a solution to this rather tricky problem,
I used the following ingenious approach ... */
...
```

Submit your personal file by 17:00 via UvA Blackboard (http://blackboard.uva.nl).