Dou Dou Search

A simple way to learn Prolog

Nahim Medellín Torres A01700190

Context of the problem

Whenever someone says the word “virtual game” or “computer game”, the first thing that comes to mind is almost always a video game: this graphically complex challenge that has super special effects, sounds, and the whole interactive experience. But the concept of games goes so much further than that, we could talk about board games, riddles, playgrounds, and even plain-text games.

Now, when we talk about learning something, and specifically programming languages, we just *know* we will be solving tedious problems such as inverting lists, simple mathematical operations, pointless queries and so on.

But what if learning didn’t have to be necessarily boring or seemingly pointless?

Okay, to be fair with the boring exercises previously mentioned, according to James Clear in his publication *Stop Thinking and Start Doing: The Power of Practicing More* practicing is learning, but learning is not practicing. What he means is, learning without action is unproductive and ultimately not valuable at all. So why don’t we combine practicing with doing something meaningful for ourselves such as having fun?

Solution

Coding the Dou-Dou Search, an adventure game entirely created using Prolog, helped me relive some precious childhood memories while practicing and getting a deeper understanding of Prolog and Logic Programming. The aim of this document is to explain the fundamental bits of the program so anyone reading it can recreate their own, going from the most simple parts of it and building up towards the most complex ones.

Why Prolog?

Prolog was originally used for research in natural language processing, although what gave prolog the popularity it has now is the artificial intelligence community because it allows for more rapid development and prototyping than most languages because it is semantically simple. Prolog relies on logical relationships to determine whether or not a statement is true, and if so what variable bindings make them true. Because of this, Prolog is also a super nice language to understand better declarative programming.

Facts

These are the simplest Prolog predicates and ressemble the records in a relational database, and that can be queried just like them. Facts are the backbone of any prolog program since they store the data that will be used.

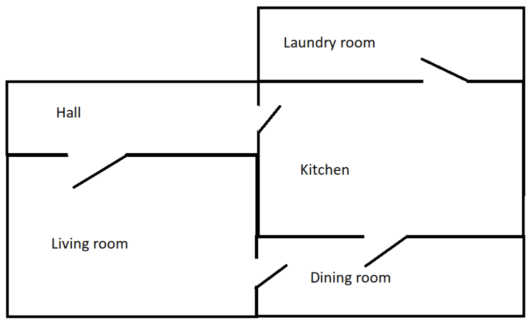


Figure 1

We can start programming the Dou-Dou search by stating facts about the map shown in Figure 1. Facts are written the following way:

```
room(kitchen).
room('laundry room').
room(hall).
room('dining room').
room('living room').
```

Notice that all the rooms names are written in uppercase letters because they are constants, whereas variables should start with capital letters. Spaces have to be treated as strings which we enclose in single quotation marks. Let's add some more facts:

```
furniture(desk).
furniture('washing machine').
furniture(table).
furniture(tv).

edible(apple).
edible(crackers).

tastes_yucky(zucchini).
```

So far, the facts have been one-argument, but we can add new facts with two arguments to model relationships. These facts are written in the manner relationship(A,B). where A has the relationship with B, but not the other way around (unless clearly stated as relationship(B,A).). For instance, we can take the following facts:

```
location(desk,'living room').
location(crackers,desk).
location(apple,kitchen).
location(flashlight,desk).
```

For the first one, we can say the desk is located in the living room, but the living room is not located in the desk.

Now we need to establish the connection between rooms. Just like in real life, we want to be able to go back and forth between rooms as we move through the game. One way to approach this is to create a relationship called room(A,B) to define the connection, and then define the opposite room(B,A) to make a two-way door. However, there is a simplest way to do this. We will start by declaring the one-way doors. The next step will be explained in the following section.

```
door(hall, kitchen).
door(hall, 'living room').
door('dining room', kitchen).
door(kitchen, 'laundry room').
door('living room', 'dining room').
```

Rules and Queries

As previously said, Prolog is a declarative language based on logic rules. This allows us to do queries using this logic. We could query whether the kitchen is a room in the form

```
?- room(kitchen).
true
```

and Prolog will consult the facts database and try and look if there is a fact that matches the query. Since we have declared that the kitchen is in fact a room, we will get a “true”. On the other hand, if we consulted room(bathroom). we would get false as the response.

We could also use variables so that we can get in return the piece(s) of information that would successfully unify with the clause. Variables are written with the first letter being capitalized. If there are many facts that allow the variable to unify, then Prolog will list them for you.

```
?- location(Thing, kitchen).
Thing = apple ;
Thing = table ;
false
```

Thing is the variable (as capitalized), and we get that both apple and table are, in fact (pun intended) located in the kitchen. We get a “false” at the very end because there were no other facts that could unify with the variable.

Now, to address the problem we currently have with one-way doors, we can make use of this queries along with Rules. A rule is composed of one or several queries that must be met as defined by it.

```
 connect(X,Y):-
    door(X,Y)
.
```

This rule reads: X is connected to Y if X has a door that leads to Y. And then we would have to declare the rule the other way around:

```
  connect(X,Y):-
     door(Y,X)
.
```

Which is read as: X is connected to Y if Y has a door that leads to X.

So if we were to write this in the listener:

```
?- connect(kitchen, hall).
```

The first thing that would happen is

1. Prolog will go to the first declared rule of connect(X,Y)
2. X would take the value of kitchen
3. Y would take the value of hall
4. PL will check if there is a fact that declares room(kitchen, hall)
    a. There is no rule declaring that, only room(hall,kitchen).
5. Upon realising that there is no fact that states what is asked, Prolog will move on to the second declaration of connect(X,Y) in an attempt to get a true response.
6. X would take the value of kitchen
7. Y would take the value of hall
8. PL will check if there is a fact that declares room(hall, kitchen) because now the clause we asked to be fulfilled is room(Y,X)
9. Since there is a fact room(hall,kitchen) we will get a true in return

There is a simpler way of finding out whether two rooms are connected or not. We do so by making up the rule with two clauses and connecting them through logic operators and/or. We use a comma (,) for AND and a semicolon (;) for OR. This way we can merge the two separate rules connect(X,Y) into a single one:

```
connect(X,Y):-
    door(Y,X);
    door(X,Y)
.
```

in which we are now stating that two rooms, for instance, kitchen[X] and hall [Y], are connected if there is a fact that states door(kitchen,hall) OR door(hall,kitchen).

Recursion

Recursion is defined as the ability for a unit of code to call itself repeatedly as necessary. In prolog we achieve recursion by declaring a predicate that contains a goal that refers to itself.

When we declare a recursive rule, each call to the rule will have its unique set of variables. To further explain this, let's take a look at this new rule called contains(Thing,Here) to check if there is a certain object inside another object inside a room.

```
contains(Thing,Here):-
    location(Thing,Here)
.

contains(Thing,Here):-
    location(Thing,X),
    contains(X,Here)
.
```

For instance if we are looking for the flashlight (which is in the desk) in the living room we would call contains(flashlight, ‘living room’). and what will happen is:

1. The first declaration of contains(Thing,Here). will be evaluated, Thing is now flashlight, Here is now ‘living room’
2. A fact location(flashlight, ‘living room’) will be looked for, for it only to fail
3. The second declaration of contains(Thing,Here). will be evaluated, Thing is now flashlight, Here is now ‘living room’
4. Now the fact location(flashlight,X) will be evaluated, leaving pending the second part of the rule which calls for contains() again.
5. X will unify with desk
6. Now it moves on to the second part of the rule, the contains(X, Here) part. Now X is desk so what is queried is contains(desk, ‘living room’).
7. The first declaration of contains(Thing,Here). will be evaluated, Thing is now desk, Here is now ‘living room’.
8. A fact location(desk, ‘living room’) will be looked for, for it to succeed.
9. We get a true in return.

Data Managing

While there are programs that work just fine with static facts, a game has to change aspects of its data to work properly and simulate events. Prolog provides built-in predicates to manipulate dynamic facts in the logicbase directly during runtime. The three predicates that are used in the Dou-Dou Search are asserta(X) which asserts as clause as the first one for a dynamic predicate, assertz(X) which is similar to the previous one but for the last clause, and retract(X) which entirely removes the clause X from the logicbase.

Now that we know we can dynamically change facts, we will get into the most important features of the game which are taking things and moving through rooms. Let’s go with the latter one.

At the beginning of the game we set the player in a room using a dynamic fact:

```
assertz(here('living room')).
```

Then, to move to another room we will break the problem into 3 smaller rules so that it’s easier to understand them. These are can_go, goto, and move.

```
can_go(Place):-
    here(X),
    connect(X, Place)
.
can_go(_):-
    write('There is no way for you to get there from here.'), nl,
    fail
.

goto(Place):-
    can_go(Place),
    move(Place),
    look
.

move(Place):-
    retract(here(_)),
    asserta(here(Place))
.
```

We have already reviewed how rules work, so the first two rules should be easy to understand, but we will go step-by-step one last time.

Let’s suppose we are in the living room (as stated in the initial fact) and we want to move to the hall (which we already know is legal, but prolog does not). So we call goto(hall).

1. Place is now hall, so Prolog finds the goto(Place) and tries can_go(hall)
2. The first declaration of can_go(Place) is evaluated unifying X to the fact here(‘living room’).
3. The second clause of can_go(Place) is checked, which is connect(‘living room’, hall). It returns true because it exists.
    a. Should it have returned false, the next declaration of can_go would have been evaluated and accordingly displayed the “There is no way for you to get there from here.' message.
4. Because the first clause of goto(Place) was true, we move onto move(Place). with Place being hall
5. Once we are in move(hall) we retract the existing fact of here(_) using an underscore to tell Prolog that we do not care what is in there, just do what it is being told to (in this case, getting rid of that fact).
6. Then we use asserta(here(hall)) to declare that now the player is in the hall.
    a. NOTE. We could use assertz since the last fact was already retracted, but just to super duper double-proof it we use asserta to make sure this new fact is the first one that will be evaluated the next time here(X) is checked.

Finally, we use the rule look to look around the room. Why don’t you give it a shot? This rule should display the current room, a list of the things that can be seen, and the list of places you can move to, use a separate rules to list the things and the connections. The solution for this can be found from line 156 to line 175 on the doudousearch.pl file in the repository.

Finishing up the beta version

Up to this point we have the backbone of a simple plain-text game in Prolog. More rules can be implemented to provide a richer experience both for the user and the programmer, such as turning on and off the flashlight, eating stuff, taking things from places and leaving them somewhere else, checking the inventory. All of these rules are implemented in the doudousearch.pl file for anyone to both use and get inspiration from according to the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

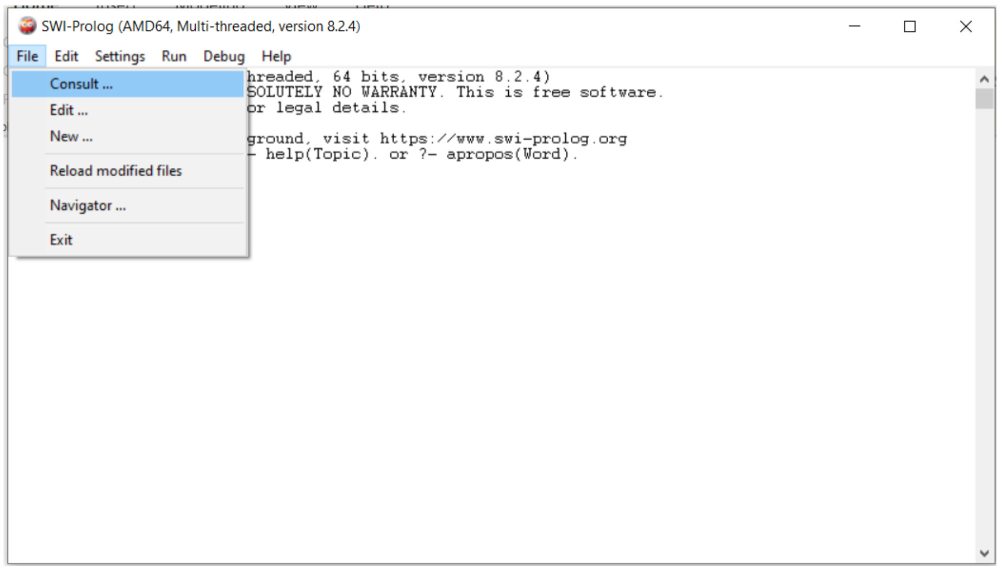
One step further

To provide a much more richer experience for the end-user I suggest you learn about Natural language in Prolog. It goes much further than was intented for you to learn with this project, but it definitely worth taking a look at. I recommend taking a look at this Chatbot project made for Prolog which explains into detail how to do basic NLP, which was also used for the development of Dou Dou Search.

Test the Dou Dou Search

To test the Dou Dou Search yourself (both in natural language and each rule separately) you have to install your preferred Prolog IDE. In my case, I recommend the SWI-Prolog one.

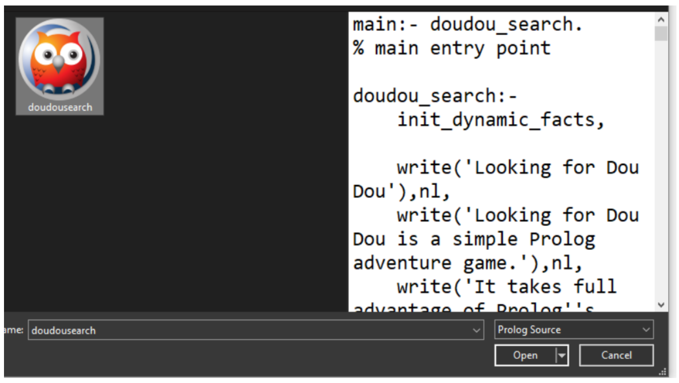
Once installed, all you need to do is open SWI-Prolog, go to File > Consult



```
hreaded, 64 bits, version 8.2.4)
SOLUTELY NO WARRANTY. This is free software.
or legal details.

ground, visit https://www.swi-prolog.org
- help(Topic). or ?- apropos(Word).
```

Select the doudousearch.pl file and open it



```
main:- doudou_search.
% main entry point

doudou_search:-
    init_dynamic_facts,

    write('Looking for Dou
Dou'),nl,
    write('Looking for Dou
Dou is a simple Prolog
adventure game.'),nl,
    write('It takes full
advantage of Prolog''s
```

Then type main. and start playing!

Conclusion

Making both the Dou Dou Search and this document was an enriching experience that I enjoyed from beginning to end. I made my best effort to explain the main aspects of Prolog in a natural, understandable way so that almost anyone with programming experience can learn Prolog and know that there is so much more to it than just family trees. Sure, solving data structures problems and inverting lists is super fun (not) but there is nothing that is more fun than playing/making games. So consider this a kind invitation to make use of Prolog's silent power. And to use this as a quick guide to study for your finals. Go find Dou Dou before it's too late!