COSC343: CORE PROLOG TECHNIQUES FOR AI

LECTURE 5

In today's lecture

Last lecture, we looked at how Prolog works (i.e. its search strategy).
In this lecture, we'll look at Prolog more practically, from the programmer's point of view.

- ♢ Prolog as a declarative language
- ♢ Non-logical predicates
- ♢ Some common list-processing tricks

We'll also look at how to implement your own search strategies in Prolog.

Prolog as a declarative language

In a **procedural** language (e.g. Java), you run a function on some input data, and it returns some output data as a result.

Prolog is a **declarative** language: there's no explicit input and output. But you can get the effect of input and output by using variables in your query. For instance, assume we have a simple database:

```
mother_of(liz, charlie).
```

By changing the pattern of variables in our query, we can get Prolog to 'return' the child given the mother, or the mother given the child:

```
?- mother_of(liz, X).

?- mother_of(X, charlie).
```

Commenting Prolog predicates

Comments are introduced by %.

- Comments on a predicate often state what relationship it implements, rather than what procedure it defines.

- Bad:
  ```
  % mother(X,Y) takes a mother X and returns a child Y.
  ```

- Good:
  ```
  % mother(X,Y) is true if X is the mother of Y.
  ```

Building material into the head of a rule

Prolog's declarative structure means you never need unification statements in the body of a rule.

If you're thinking procedurally:

```
starts_with_a(List) :-
     List = [a|Rest].
```

If you're thinking in Prolog:

```
starts_with_a([a|Rest]).
```

Non-logical predicates: write, assert, retract

Not all Prolog clauses are completely declarative. Some have **side-effects**.

- Printing:

```
?- writeln('Ali').
Ali
Yes
```

- `assert` and `retract` have side-effects on the database:

```
?- genius(ali).
No
?- assert(genius(ali)).
Yes
?- genius(ali).
Yes
?- retract(genius(ali)).
Yes
```

Non-logical predicates: maths functions

Another non-logical predicate is needed for mathematical computation.

Mathematical computations don't work if you use term unification:

```
?- X = 2+2.
X = 2+2
```

The special infix operator `is` is used for this kind of computation:

```
?- X is 2+2.
X = 4
```

However, `is` only accepts variables in the first argument position:

```
?- X is Y.
ERROR: Arguments are not sufficiently instantiated
```

The cut

The **cut** symbol (`!`) is an indication to Prolog that it never needs to backtrack past the current point.

For example: if we load the following predicate:

```
f(X,0) :- X < 3, !.
f(X,2).
```

We get the following behaviour:

```
?- f(1, X).
X = 0 ;
No
```

Green cuts are just used for efficiency.
Red cuts actually change the behaviour of a predicate.

Non-logical predicates: negation

Prolog implements a form of explicit negation, using the predicate `not/1` (or the operator \+). Assume we have a database with a single fact in it:

```
fish(jim)
```

Now:

```
?- not(fish(jim)).
No
?- not(fish(bill)).
Yes
```

Here's a reimplementation of `not/1`:

```
not*(P) :-
  P, !, fail.
not*(P).
```

SWI error messages

When consulting a Prolog file, there are two common error messages:

```
Warning:  Singleton variable (...)
```

- You get this if one of your clauses has just one instance of a variable.

  ```
  loves(harry, X).
  ```

  To get rid of the problem, use the **anonymous variable** _.

  ```
  loves(harry, _).
  ```

- Singleton variable often indicate bugs (e.g. misspelled variables). So it's useful to get rid of them.

```
Warning:  clauses of (..)  are not together in the source file
```

- I never understood why this is a problem :-)

List processing: recap

Last lecture, we looked at the member* predicate.

```
%member(Item, List) succeeds if Item is a member of List.

member*(X, [X|Rest]).    %base case
member*(X, [_|Rest]) :-  %recursive case
     member*(X, Rest).

?- trace, member*(b, [a, b, c])
   Call: (8) member* (b, [a, b, c]) ?
   Call: (9) member* (b, [b, c]) ?
   Exit: (9) member* (b, [b, c]) ?
   Exit: (8) member* (b, [a, b, c]) ?

Yes
```

Experimenting with different variable patterns

```
?- member*(X, [a,b]). <-- returns each member of a list in turn
X = a ;
X = b ;
No

?- member*(a,[X,b]).  <-- 'creates' a list containing 'a'
X = a ;
No

?- member*(X,Y).      <-- 'creates' an infinite sequence of lists.
X = _G303
Y = [_G303|_G375] ;
X = _G303
Y = [_G374, _G303|_G381] ;
(...)
```

A definition for 'append'

```
%append(L1, L2, L3) succeeds if L3 is the list which results
%from appending L1 to L2.

append*([], List, List).                        %base case

append*([X|Rest1], List, [X|Rest2]) :-  %recursive case
     append*(Rest1, List, Rest2).
```

This predicate basically works by stripping elements off the first list 'on the way down' to the base case (when it's empty) and then adding them onto the second list 'on the way back out' of the recursion.

A predicate for processing each element in a list

```
%Process_list(List, New_list) succeeds if New_list is a list
%containing the results of doing [some operation] on the
%elements of List.

process_list([],[]).
process_list([First|Rest],[p/First|New_rest]) :-
process_list(Rest, New_rest).
```

This predicate also works by building the result list 'on the way back out' of the recursion.

Implementing state-space search in Prolog

Although Prolog implements its own state-space search, it is useful to build *our own* search function as a Prolog program.

- The main reason is to allow us to implement different **search strategies**. (Remember that Prolog's own search strategy is always depth-first.)

What we need for this:

- A method for representing a state space
- A method for specifying a goal state
- A `search` predicate, which takes a start state as 'input' and computes a sequence of actions as 'output'.

Representing a state space and a goal state

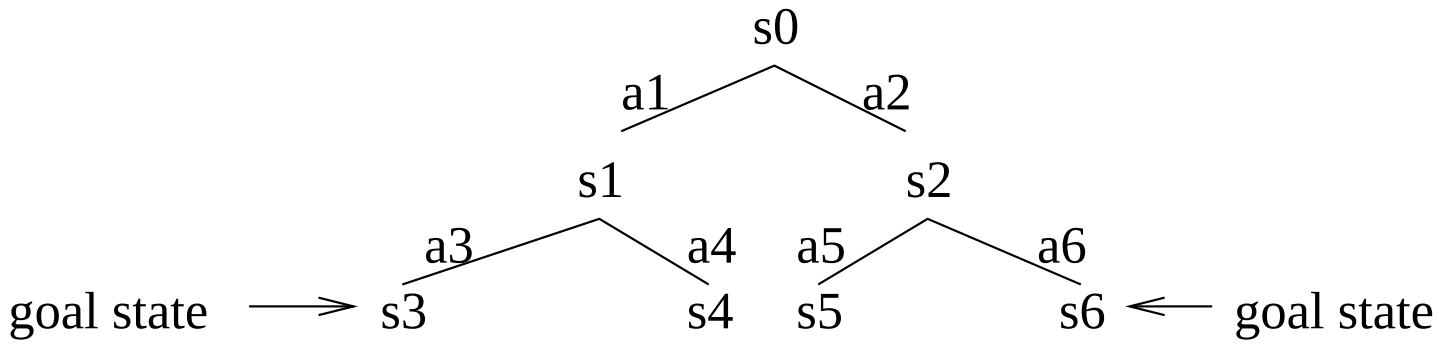
Recall: the state-space graph is defined implicitly,
by a **successor function**.

- The successor function takes a state $S1$ and returns a set of action-state pairs (where each state is the consequence of doing that action in $S1$).
- So: what we need to do is to write a Prolog predicate which defines a successor function.
- We can call the predicate `neighbours/2`.

We can also define a predicate called `is_goal/1`, to represent one or more goal states.

An example Prolog state space and goal state

Let's take a simple state space, with no cycles:



We can represent this as the following Prolog database:

```
neighbours(s0, [[a1,s1], [a2,s2]]).
neighbours(s1, [[a3,s3], [a4,s4]]).
neighbours(s2, [[a5,s5], [a6,s6]]).

is_goal(s3).
is_goal(s6).
```

Representing the search tree

Recall: a search tree is made up of **nodes**.

Let's represent a node as a list of two elements: `[State,Action_history]`. (Where `Action_history` holds the sequence of actions by which `State` was reached from the start state.)

- Q: What will the **goal nodes** in our search tree be?

The **fringe** will be a list of nodes.

The top-level search predicate

Assume our top-level search predicate implements a breadth-first search. Assume it returns the sequence of actions needed to get to a goal state:

```
?-  top_level_search(s0, X).

X = [a1, a3] ;
X = [a2, a6] ;
No
```

The top-level predicate needs to create a node for the start state, and call a *recursive* search predicate with this single node on the fringe.

```
top_level_search(Start_state, Action_sequence) :-
   search([[Start_state,[]]], Goal_node),
   Goal_node = [_,Action_history],
   reverse(Action_history, Action_sequence).
```

The recursive search predicate

```
%search(Fringe, Goal_node) succeeds if Goal_node is reachable
%from Fringe.

search([Node|_], Node) :-
    Node = [State, _],
    is_goal(State).

search([Node|Rest_of_fringe], Goal_node) :-
    Node = [State,Action_history],
    write('Expanding state '), writeln(State),
    neighbours(State, Neighbour_list),
    create_nodes(Action_history, Neighbour_list, Child_nodes),
    append(Rest_of_fringe, Child_nodes, New_fringe),
    search(New_fringe, Goal_node).
```

The create_nodes predicate

create_nodes/2 is a list-processing predicate, with the same form as process_list/2. It basically creates a node for each element of the list returned by neighbours/2.

```
create_nodes(_, [], []).
create_nodes(Action_history, [Neighbour|Rest_neighbs],
                             [Node|Rest_nodes]) :-
        Neighbour = [Action,State],
        Node = [State,[Action|Action_history]],
        create_nodes(Action_history, Rest_neighbs, Rest_nodes).
```

Testing the search predicate

```
?- top_level_search(s0, X).
Expanding state s0
Expanding state s1
Expanding state s2

X = [a1, a3] ;              <---- First solution
Expanding state s3
Expanding state s4
Expanding state s5

X = [a2, a6] ;              <---- Second solution
Expanding state s6

No                          <---- No more solutions
```

Alternative search strategies

Here's the line which determines the breadth-first search strategy:

```
append(Rest_of_fringe, Child_nodes, New_fringe),
```

Some questions:

- How would you implement a depth-first strategy?

- How would you implement a uniform-cost strategy?

Summary and reading

Prolog as a declarative language

More list-processing techniques

Non-logical predicates in Prolog: writing, asserting, retracting, maths...

Representing a state space in Prolog

Building a tree-searcher in Prolog

There was no reading for this lecture.
Reading for next lecture: AIMA Ch4 Sections 1 and 2.