Deerwalk Institute Of Technology

deerwalk
DWIT College

Lab 3: Problem Solving in Prolog

(Artificial Intelligence)

Submitted by:

Name: Shiva Tripathi

Roll No: 532

Batch: 2019

Section: A

Submitted to:

Birodh Rijal

Practical IV - Prolog Basics-2 Submission Deadline: TBA

1 Problem Solving

A. Given the following directed graph:

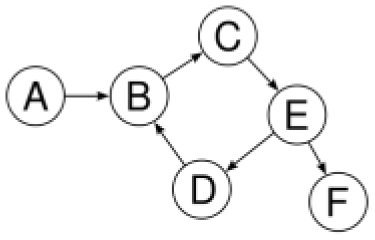

Figure 1: A directed graph with six nodes

Write a prolog program that would check if there is a path from any node X to any other node Y. Encode your database of facts like: edge('A','B'). , edge('E','F'). and so on. Now wrie a rule for the predicate path(X,Y) that is *true* if there is a path from node represented by the variable X to the path represented by variable Y. You might have guessed that this requires the rule to be recursive. As you have to do while writing recursive function, think of the base case and the recursive step.

```
edge(a,b).
edge(b,c).
edge(c,e).
edge(e,d).
edge(d,b).
edge(e,f).

toedge(F_node,L_node):-edge(F_node,L_node).
toedge(F_node,L_node):-edge(F_node,SomeNode), edge(SomeNode,L_node).
```

```
?- consult('graph.pl').
true.

?- toedge(a,c).
true.
```

B. Using the concept of recursion add a rule predecessor(X,Y) to the family tree in the third lab that is true if X is the predecessor of Y (i.e. X comes before Y in the parent relation).

```
Predecessor(X,Y):-parent(X,Y).

Predecessor(X,Y):-parent(X,W),predecessor(W,Y).
```

C. The classic monkey and banana problem follows. Model this problem as a production system. There is a monkey at the door into a room. In the middle of the room a banana is hanging from the ceiling. The monkey is hungry and wants to get the banana, but he cannot stretch high enough from the floor. At the window of the room there is a box the monkey may use.
The monkey can perform the following actions: (a) Walk on the floor; (b) Climb the box (if it is already at the box); (c) Push the box around (if it is already at the box); (d) Grasp the banana (if standing on the box directly under the banana).
First we need to find the state representation. The current state in this problem is defined by the position of the objects. We need to store: (i) monkey's location in the room; (ii) whether monkey is on the floor or on the box; (iii) the location of the box; (iv) if the monkey has the banana or not. We will define a predicate state/4 for this task. The 4 at the end is called the arity of the predicate (basdically it is the number of arguments it takes).
Initial State: Monkey is at the door. Monkey is on floor. Box is at window. Monkey does not have banana. In prolog we can write it as, state(atdoor, onfloor, atwindow, hasnot). Goal state: state(, , , has). Here the underscore denotes the 'don't care' values. This is to say, all we care is if the monkey has the banana or not. For the problem to be solved the only thing we need to check is if the monkey has the banana or not.
Now, we will write the rules for the valid moves. We will define a predicate do/3 for this. The first argument is the current state, second argument is the action performed and the third is the new state resulting from carrying out the action.

(a) Grab the banana
```
do(state(middle, onbox, middle, hasnot), grab, state(middle, onbox, middle,
has)).
```

(b) Climb the box
```
do(state(L, onfloor, L, Banana), climb, state(L, onbox, L, Banana)).
```

(c) Push box from L1 to L2
```
do(state(L1, onfloor, L1, Banana), push(L1, L2), state(L2, onfloor, L2, Banana)).
```

(d) Walk from L1 to L2
```
do(state(L1, onfloor, Box, Banana), walk, state(L2, onfloor, Box, Banana)).
```

Finally we now define the main question our program will pose: \Can the monkey in some initial state get the banana?". For that we define a predicate canget(State). Here State is the current state. Now, for any state in which the monkey already has the banana this predicate is true. So we write:
canget(state(, , , has)).
In other cases one or more moves are necessary. The monkey can get the banana in any state (State1) if there is some move (Move) from State1 to some state (State2), such that the monkey can get the banana in State2 (in zero or more moves).
canget(State1) :- move(State1, Move, State2), canget(State2).
Note that the canget/1 predicate is recursive. Give these queries from the prolog prompt and check:
?- canget(state(atwindow, onfloor, atwindow, has)).
?- canget(state(atdoor, onfloor, atwindow, hasnot)).
?- canget(state(atwindow, onbox, atwindow, hasnot)).

```
do(state(middle, onbox, middle, hasnot), grab, state(middle, onbox, middle,has)).

do(state(L, onfloor, L, Banana), climb, state(L, onbox, L, Banana)).

do(state(L1, onfloor, L1, Banana), push(L1, L2), state(L2, onfloor, L2, Banana)).

do(state(L1, onfloor, Box, Banana), walk, state(L2, onfloor, Box, Banana)).

canget(state(_,_,_, has)).

canget(State1) :- move(State1, Move, State2), canget(State2).
```

```
?- canget(state(atwindow, onfloor, atwindow, has)).

true .


?- canget(state(atdoor, onfloor, atwindow, hasnot)).

true .


?- canget(state(atwindow, onbox, atwindow, hasnot)).

false.
```

D. Towers of Hanoi
To move n discs from peg A to peg C, the recursive algorithms is:

move n1 discs from A to B. This leaves disc n alone on peg A
move disc n from A to C
move n1 discs from B to C so they sit on disc n
In prolog code:
move(1,X,Y,) :- write('Move top disk from '), write(X), write(' to '), write(Y),
nl.
move(N,X,Y,Z) :- N>1, M is N-1, move(M,X,Z,Y), move(1,X,Y,), move(M,Z,Y,X). hanoi(N)
:- move(N,'A','C','B').
We now can give the following query to this program from the prolog prompt: ?- hanoi(3).
There are few things to note here.
write is a built-in predicate. Think of it as the printf function. nl prints a newline.
X=Y-1 is written as X is Y-1 in prolog.
In hanoi(N), N is the number of discs and A, B and C are the pegs (hence constants and therefore in single quotes).

```
move(1,X,Y,_) :-
    write('Move top disk from '),
    write(X),
    write(' to '),
    write(Y),
    nl.
move(N,X,Y,Z) :-
    N>1,
    M is N-1,
    move(M,X,Z,Y),
    move(1,X,Y,_),
    move(M,Z,Y,X).
```

```
?- move(3,left,right,center).
Move top disk from left to right
Move top disk from left to center
Move top disk from right to center
Move top disk from left to right
Move top disk from center to left
Move top disk from center to right
Move top disk from left to right
true .
```