Prolog Programming Assignment #2: State Space Problem Solving

What's It All About?

This assignment leads you through the development of a state space problem solver for the Towers of Hannoi problem in Prolog. The search is blind, without an sort of heuristic guidance. Consequently, this assignment should leave you wanting more! That is, it should entice you to want to know something more about state space search.

Tasks

1. Thinking carefully, do the first task. Then, working carefully within a nice text editor, and with a good Prolog interpreter, do the remaining eight tasks.
2. Craft a nicely structured solution document that contains representations of each of the seven tasks for which you are actually asked to post something, by which I mean incorporate something into your solution document (Tasks 3 throuth 9). Moreover, be sure to title the document, and place a "learning abstract" just after the title, before presenting your work on each of the seven tasks for which you are asked to incorporate something into your solution document.
3. Post your solution document to your web work site.

Task 1: Problem Contemplation - Towers of Hanoi

This programming challenge affords you an opportunity to implement a state space problem solver for the Towers of Hanoi problem. Please review the problem statement, and then contemplate the representation that was presented in class.

Problem Statement - Towers of Hanoi

The three peg / three tower problem: Three pegs/towers. Three disks large (L), medium (M), small (S). The disks are place on the pegs subject to the constraint that a larger disk "cannot appear" on top of a smaller disk. A move consists of transferring a disk, the top one, from one peg to another, placing it on top of whatever disks may be present. The task is to transfer all of the pegs from the first peg to the third peg.

The four peg / three tower problem: Three pegs/towers. Four disks huge (H), large (L), medium (M), small (S). The disks are place on the pegs subject to the constraint that a larger disk "cannot appear" on top of a smaller disk. A move consists of transferring a disk, the top one, from one peg to another, placing it on top of whatever disks may be present. The task is to transfer all of the pegs from the first peg to the third peg.

The five peg / three tower problem: Three pegs/towers. Five disks huge (H), large (L), medium (M), small (S), tiny (T). The disks are place on the pegs subject to the constraint that a larger disk "cannot appear" on top

of a smaller disk. A move consists of transferring a disk, the top one, from one peg to another, placing it on top of whatever disks may be present. The task is to transfer all of the pegs from the first peg to the third peg.

State Space Representation - Towers of Hanoi

For the three disk problem, represent the three disks by symbols L (large) and M (medium) and S (small). Represent the three pegs as lists, imagining the disks arranged from left to right in increasing order of size.

Then ...

- I = {`((S M L) () () )`}
- G = {`(() () (S M L) )`}
- O = {`M12, M13, M21, M23, M31, M32`}, where
  - M12 - move a disk from peg 1 to peg 2
  - M13 - move a disk from peg 1 to peg 3
  - M21 - move a disk from peg 2 to peg 1
  - M23 - move a disk from peg 2 to peg 3
  - M31 - move a disk from peg 3 to peg 1
  - M32 - move a disk from peg 3 to peg 2

One possible state space solution:

M13 ⇒ M12 ⇒ M32 ⇒ M13 ⇒ M21 ⇒ M23 ⇒ M13

For the four disk problem, represent the four disks by symbols H (huge) and L (large) and M (medium) and S (small). Represent the three pegs as lists, imagining the disks arranged from left to right in increasing order of size. Adjust the initial state and the goal state appropriately.

For the five disk problem, represent the five disks by symbols H (huge) and L (large) and M (medium) and S (small) and T (tiny). Represent the three pegs as lists, imagining the disks arranged from left to right in increasing order of size. Adjust the initial state and the goal state appropriately.

Task 2: Code Contemplation

Please review the Missionaries and Cannibals state space problem solving program that was presented in Lesson 7. Then, contemplate the following *unrefined* state space problem solving program for the Towers of Hanoi problem. In subsequent tasks, you will be asked to refine and demo this code.

When you are ready, place this text into a file called `toh.pro`. (Be sure **not** to place the "redacted code" tags in your file.) Also, place the `inspector.pro` file, provided as an appendix to this programming assignment, in your computational world as a sibling to the `toh.pro` file. Then, load this code into a Prolog process, just to be sure that everything is in order before you commence with the subsequent tasks.

```
% ------------------------------------------------------------------------
% ------------------------------------------------------------------------
% --- File: towers_of_hanoi.pro
% --- Line: Program to solve the Towers of Hanoi problem
% ------------------------------------------------------------------------

:- consult('inspector.pro').

% ------------------------------------------------------------------------
% --- make_move(S,T,SSO) :: Make a move from state S to state T by SSO

make_move(TowersBeforeMove,TowersAfterMove,m12) :-
  m12(TowersBeforeMove,TowersAfterMove).
make_move(TowersBeforeMove,TowersAfterMove,m13) :-
  m13(TowersBeforeMove,TowersAfterMove).
make_move(TowersBeforeMove,TowersAfterMove,m21) :-
  m21(TowersBeforeMove,TowersAfterMove).
make_move(TowersBeforeMove,TowersAfterMove,m23) :-
  m23(TowersBeforeMove,TowersAfterMove).
make_move(TowersBeforeMove,TowersAfterMove,m31) :-
  m31(TowersBeforeMove,TowersAfterMove).
make_move(TowersBeforeMove,TowersAfterMove,m32) :-
  m32(TowersBeforeMove,TowersAfterMove).

<<redacted: the six state space operators>>

% ------------------------------------------------------------------------
% --- valid_state(S) :: S is a valid state

<<redacted: valid_state>>

% ------------------------------------------------------------------------
% --- solve(Start,Solution) :: succeeds if Solution represents a path
% --- from the start state to the goal state.

solve :-
  extend_path([[[s,m,l],[],[]]],[],Solution),
  write_solution(Solution).

extend_path(PathSoFar,SolutionSoFar,Solution) :-
  PathSoFar = [[[],[],[s,m,l]]|_],
  showr('PathSoFar',PathSoFar),
  showr('SolutionSoFar',SolutionSoFar),
  Solution = SolutionSoFar.
extend_path(PathSoFar,SolutionSoFar,Solution) :-
PathSoFar = [CurrentState|_],
  showr('PathSoFar',PathSoFar),
  make_move(CurrentState,NextState,Move),
  show('Move',Move),
  show('NextState',NextState),
  not(member(NextState,PathSoFar)),
  valid_state(NextState),
  Path = [NextState|PathSoFar],
  Soln = [Move|SolutionSoFar],
```

```
  extend_path(Path,Soln,Solution).

% ----------------------------------------------------------------------
% --- write_sequence_reversed(S) :: Write the sequence, given by S,
% --- expanding the  tokens into meaningful strings.

write_solution(S) :-
  nl, write('Solution ...'), nl, nl,
  reverse(S,R),
  write_sequence(R),nl.

<<redacted: write_sequence>>

% ----------------------------------------------------------------------
% --- Unit test programs

<<redacted: the unit test programs>>
```

Task 3: One Move Predicate and a Unit Test

For this task you are given some code, and simply asked to enter it and run it. The code consists of the implementation of a state space operator, and a unit test program for the operator. You are also provided with a unit test demo.

Please note that this state space operator, as well as the other five, simply moves a disk from one peg to another, whether or not the move is "legal".

State Space Operator Implementation

Please add the following code, which implements the state space operator to move a disk from peg 1 to peg 2, `m12`, to your `toh.pro` file.

```
m12([Tower1Before,Tower2Before,Tower3],[Tower1After,Tower2After,Tower3]) :-
  Tower1Before = [H|T],
  Tower1After = T,
  Tower2Before = L,
  Tower2After = [H|L].
```

Unit Test Code

Please add the following code, which performs a unit test for the `m12` predicate, to your `toh.pro` file.

```
test__m12 :-
  write('Testing:  move_m12\n'),
```

```
  TowersBefore = [[t,s,m,l,h],[],[]],
  trace('','TowersBefore',TowersBefore),
  m12(TowersBefore,TowersAfter),
  trace('','TowersAfter',TowersAfter).
```

Unit Test Demo

Please run the unit test. If it works, great! Otherwise, fix what needs to be fixed.

```
bash-3.2$ swipl
<<redacted>>

?- consult('toh.pro').
%  inspector.pro compiled 0.00 sec, 7 clauses
% toh.pro compiled 0.00 sec, 56 clauses
true.

?- test__m12.
Testing:  move_m12
TowersBefore = [[t,s,m,l,h],[],[]]
TowersAfter = [[s,m,l,h],[t],[]]
true.

?-
```

What to Post?

Please post the code that implements the state space operator, the unit test code, and the unit test demo, being sure to do so in a clear and obvious manner.

Task 4: The Remaining Five Move Predicates and a Unit Tests

Please add code to implement the remaining 5 state space operators (`m13` and `m21` and `m23` and `m31` and `m32`). Please add a unit test program, analogous to that provided in the previous task, to test each of the five state space operators that you are asked to write for this task.

After all of the unit tests confirm that your code is good for the state space operators, perform a demo that runs all **six** unit test programs, thus assuring that all **six** state space operators are performing as they should.

What to Post?

For this task, please post (1) the code for all **six** state space operators, (2) the code for all **six** unit test programs, and (3) the demo in which all **six** unit test programs are run.

Task 5: Valid State Predicate and Unit Test

In this task you are asked to write a predicate to check whether or not a state in the Towers of Hanoi problem is valid. You are also provided with a unit test program to use in assuring that your code is sound.

Please review the given demo. Then please review the given test program, which are you required to use to assure that your predicate to check the validity of a state is sound.

Then write the predicate of one parameter called `valid_state` to do what needs to be done, that is, to check to see that each of the three towers is properly formed. Once you have written the predicate, test it with the given unit tester. If it works, great! If not, fix your code for the `valid_state` predicate.

Unit Test Program Demo

```
?- test__valid_state.
Testing:  valid_state
[[l,t,s,m,h],[],[]] is invalid.
[[t,s,m,l,h],[],[]] is valid.
[[],[h,t,s,m],[l]] is invalid.
[[],[t,s,m,h],[l]] is valid.
[[],[h],[l,m,s,t]] is invalid.
[[],[h],[t,s,m,l]] is valid.
true

?-
```

Unit Test Program

```
test__valid_state :-
  write('Testing:  valid_state\n'),
  test__vs([[l,t,s,m,h],[],[]]),
  test__vs([[t,s,m,l,h],[],[]]),
  test__vs([[],[h,t,s,m],[l]]),
  test__vs([[],[t,s,m,h],[l]]),
  test__vs([[],[h],[l,m,s,t]]),
  test__vs([[],[h],[t,s,m,l]]).

test__vs(S) :-
  valid_state(S),
  write(S), write(' is valid.'), nl.
test__vs(S) :-
  write(S), write(' is invalid.'), nl.
```

What to Post?

Post (1) your code for the `valid_state` predicate, (2) my unit test program code, and (3) your unit test program demo.

Task 6: Defining the write_sequence predicate

Write the one parameter `write_sequence` that takes a sequence of symbols corresponding to a sequence of state space operators and writes the corresponding sequence of operator descriptions, in a manner consistent with the code and demo provided.

Unit Test Program Code

```
test__write_sequence :-
  write('First test of write_sequence ...'), nl,
  write_sequence([m31,m12,m13,m21]),
  write('Second test of write_sequence ...'), nl,
  write_sequence([m13,m12,m32,m13,m21,m23,m13]).
```

Unit Test Program Demo

```
?- test__write_sequence.
First test of write_sequence ...
Transfer a disk from tower 3 to tower 1.
Transfer a disk from tower 1 to tower 2.
Transfer a disk from tower 1 to tower 3.
Transfer a disk from tower 2 to tower 1.
Second test of write_sequence ...
Transfer a disk from tower 1 to tower 3.
Transfer a disk from tower 1 to tower 2.
Transfer a disk from tower 3 to tower 2.
Transfer a disk from tower 1 to tower 3.
Transfer a disk from tower 2 to tower 1.
Transfer a disk from tower 2 to tower 3.
Transfer a disk from tower 1 to tower 3.
true.

?-
```

What to Post?

Post (1) your code for the `write_sequence` predicate, (2) my unit tester code, and (3) your unit test program demo.

Task 7: Run the program to solve the 3 disk problem

Run the program to solve the three disk Towers of Hanoi problem twice:

1. With the intetermediate output displayed.
2. With just the English-like solution displayed.

Do your best to answer the following questions:

1. What was the length of your program's solution to the three disk problem?
2. What is the length of the shortest solution to the three disk problem?
3. How do you account for the discrepency?

What to Post?

(1) A demo with the intermediate output displayed, after which the English-like solution is displayed. (2) A demo with just the English-like solution displayed. (3) Answers to your three questions, contextualized by the questions.

Task 8: Run the program to solve the 4 disk problem

Run the program to solve the four disk Towers of Hanoi problem, without displaying any intermediate output. Does your program produce a solution? If so, please do your best to answer the following questions:

1. What was the length of your program's solution to the four disk problem?
2. What is the length of the shortest solution to the four disk problem?

If, on the other hand, your program does not produce a solution, please do your best to answer the following pair of questions:

1. What are the two most salient possible reasons why your program failed to produce a solution?
2. Which of the two salient reasons that you identified do you believe is responsible for your program failing to produce a solution to the problem?

What to Post?

(1) The demo. (2) Your answers to the questions.

Task 9: Review your code and archive it

Review your program to make sure that it is properly formatted. Fix it up, if need be, Then be sure to run your program to make sure that the code is still sound.

What to Post?

Please post a complete (unit tests and all) listing of your source program.

Due Date

Please complete your work on this assignment, and post your work to your web work site no later than Friday, April 22, 2022.

Appendx – Inspector Code

```
% ----------------------------------------------------------------------
% ----------------------------------------------------------------------
% --- File: inspectors.pro
% --- Line: Utilities for inspecting memory during program execution
% ----------------------------------------------------------------------

% ----------------------------------------------------------------------
% --- These two can be used to print the value of a variable, labelled
% --- in two ways, and pause for the programmer to check out the
% --- situation. The firstone is generally useful. The second one is
% --- applicable only whenthe value of the veriable is a list, and it
% --- will print the valuein reverse order which is sometimes just
% --- what is desired. The first label generally pertains to a location
% --- in the program. The second label is just the name of the variable
% --- to which the value is bound.

check(Label,Name,Value) :-
  write(Label),
  write(Name),write(' = '),
  write(Value),nl,
  read(_).

checkr(Label,Name,Value) :-
  write(Label),
  write(Name),write(' = '),
  reverse(Value,RValue),
  write(RValue),nl,
  read(_).
```

```
% ------------------------------------------------------------------------
% --- These two are like the previously described checking predicates,
% --- except that they do not do the pause.

trace(Label,Name,Value) :-
  write(Label),
  write(Name),write(' = '),
  write(Value),nl.

tracer(Label,Name,Value) :-
  write(Label),
  write(Name),write(' = '),
  reverse(Value,RValue),
  write(RValue),nl.

% ------------------------------------------------------------------------
% --- Like trace, but without the extra labelling functionality.

show(Name,Value) :-
  write(Name),write(' = '),
  write(Value),nl.

showr(Name,Value) :-
write(Name),write(' = '),
  reverse(Value,RValue),
  write(RValue),nl.
```