**20CS6033**

**Instructor: A. Ralescu**

**Fall 2022**

**Assignment 5**

**Assigned 10/24/2022**

**Due on Canvas 11/04/2022, at 11:59PM**

**50 points**

You are asked to write a program to implement the blocks’ world problem.

The paths from a start to a goal node are maintained as lists.

A state in the blocks’ world is described as a list of constraints (a nested list), such as

**[[on, a, b],[on, b, “table”], [on, c, d], [clear, c], [clear, a], [on, d, “table”]]**

The above state basically describes that “**a is on b; b is on the table; c is on d; c is clear; a is clear, d id on the table**

We want to search for the sequence of moves to arrange the blocks in the state

**[[on, d, a], [on, a, c], [on, c, b], [on, b, “table’], [clear, a]]**

|  |  |  |  |
| --- | --- | --- | --- |
| **START STATE** | |  | **GOAL STATE** |
|  |  |  | d |
|  |  |  | a |
| a | c |  | c |
| b | d |  | b |

You should define a predicate which lists all the blocks in your world: for example, your blocks world may have the blocks a, b, c, d, e, f: **blocks([a, b, c, d, e, f]).**

Then a block, say X, is a member of the list of blocks. That is, **block(X)**is defined simply by

**block(X):-**

**block(BLOCKS),** % this extracts the list BLOCKS

**member(X, BLOCKS).**

The rules for the blocks’ world are as follows:

1. A block can be clear (when no other block is on top of it), or not.
2. Any clear block can be moved from a block or from the table onto another clear block (which will no longer be clear after this moved is performed).
3. Any clear block can be moved from a block onto the table (there is always room for blocks on the table); the block from which the move was performed now becomes clear.
4. The table is infinite, in the sense that there is always room for another block on it.

Let’s refine one of these rules:

If, in a state S1, X is a clear block which is on a block Y, and Z is another clear block, then moving X from Y onto Z will change S1 to a state S2 as follows:

1. X is not on Y anymore: this means that the constraint [on, X, Y] must be substituted in S1 by [on, X, Z] leading to the intermediate state INT
2. The constraint [clear, Z] in INT must be substituted by [clear, Y] to lead to the state S2

Let us now implement this in prolog:

% move(X, Y, Z, S1, S2) holds when S2 is obtained from S1 by moving the block X from the block Y onto the block Z.

**move(X, Y, Z, S1, S2):-**

**member([clear, X], S1),** %find a clear block X in S1

**member([on, X, Y], S1), block(Y),** %find a block on which X sits

**member([clear, Z], S1), notequal(X, Z**), %find another clear block, Z

**substitute([on, X, Y], [on, X, Z], S1, INT),** %remove X from Y, place it on Z

**substitute([clear, Z], [clear, Y], INT, S2).** %Z is no longer clear; Y is now clear

%notequal(X11, X2) holds when X1 and X2 are not equal

**notequal(X,X):-!, fail.**

**notequal(\_, \_).**

% substitute(E, E1, OLD, NEW) holds when NEW is the list OLD in which E is substituted by E1. There are no duplicates in OLD or NEW.

substitute(X, Y, [X|T1], [Y|T1]).

substitute(X, Y, [H|T], [H|T1]):-

substitute(X, Y, T, T1).

You must write two more rules for the blocks’ world:

1. move from a block onto the table, and
2. move from the table onto a block

Next, define the path between two states: two states S1, and S2, are connected if there exists a move from S1 to S2:

path(S1, S2):-

move(X, Y, Z, S1, S2).

%connect: symmetric version of path

connect(S1, S2) :- path(S1, S2).

connect(S1, S2) :- path(S2, S1).

**To test your program, make up some start and goal states and include them into your program as**

start(…).

goal(…).

As the path from start to goal is being built, you will need to make sure that a newly generated state has not been visited already.

In the Romanian road map problem, this is easy: it is sufficient to require that the new city was not a member of the path built so far. However, for the blocks’ world problem, this is not sufficient, because a state must be viewed as a **set** of constraints **not as a list**.

For example, suppose that the new state is **[[on, a, b], [clear, a], [on, b, “table’]]** and that somewhere, in the path so far there is a state such as is **[[clear, a], [on, a, b], [on, b, “table’]]**. These two states are different as lists but not different as sets. **Therefore, a state is identical to any of its permutations.**

Thus, to make sure that a state has not been visited, we must **check not that the state is not in the path so far, but that any of its permutations is not there either.**

To do this, we would need to write the **notYetVisited** predicate something like:

notYetVisited(State, PathSoFar):-

**permute**(State, PermuteState),

notmember(PermuteState, PathSoFar).

**Here are some hints on how to define permute:**

1. It is obvious that it must be **recursive**, and that each recursive call must be to permute a list one element shorter.
2. This shorter list is obtained by **successively deleting each element of the list**,
3. permute the resulting shorter list, and then
4. add the deleted element in front (as the head) of the permuted version of the shorter list to obtain a permutation of the original list

Thus

permute(L, [X|P]):-

**delete** ….

permute….

The predicate delete is straightforward (complete its definition and write appropriate comment for it):

**%delete(E, OLD, NEW)…**

1st clause: the result of deleting the head of the is the tail of the list.

2nd clause: the result of deleting any other element of the list results in a list with the same head as the original list and tail obtained by deleting the element from the tail of the original list.

Include in your program knowledge base two facts stating the start and goal states:

start(…).

goal(…).

You will need to write a predicate which builds the sequence of moves. Let us call this dfs (it implements depth first search):

**%dfs(State1, Path, PathSoFar):** returns the Path from the start to the goal states.

% Trivial: if X is the goal return X as the path from X to X.  
**dfs(X, [X],\_):- goal(X).**

% else expand X by Y and find path from Y  
**dfs(X, [X|Ypath], VISITED):-  
 connected(X, Y),  
 negmember(Y, VISITED),  
 dfs(Y, FILLIN, [Y|VISITED]).**

At grading, your program will be run on another data to make sure that none of your predicates are tailored specifically to the data you chose to develop your program.

**What to turn in**

turn in the listing of the program in a file so that it can be run by me and/or the grader. Separately turn in a document in which you show the result of various runs (you can take snapshots of your screen which show the result, paste them in a word document and include comments on these results).

So, you will need to upload a zipped folder containing

1. Your program file (also including list-processing predicates).
2. The word document showing results of various runs, and your comments on these. The list of states, etc.
3. **At the top of your document write the full names of the team members and their respective contributions. If all contributed equally, just write “all contributed equally”. If not, write the percentage of the work a team member contributed. I will have to adjust the group grades by that percentage.**

**Please name your file (BlocksWorldFirstNameLastName), where FirstName and LastName are the full name of a group member.**