CSE560: Homework 9

Python: Situation Semantics

Make a forward planner that works with Situation Semantics. Your forward planner will need a depth-first theorem prover to evaluate the ‘poss’. You could use a variation of your theorem prover from earlier homeworks, but that (a) supports explicit unification, (b) supports the ‘not’ as negation as failure, but without the delaying mechanism, and (c) that searches for all solutions for a query, rather than just the first one.

Do this assignment in Python. Use the theorem prover below as your starting point. The first argument to `proveall` is the query and the second argument is the knowledge base. The third argument is optional, and is used to control the amount of indenting for print statements that show the execution of the program, which is only done if ‘ProveDebug’ is set to 1. This is used when doing negation as failure to show that you are doing an embedded proof.

```python
from hw4standard import *

ProveDebug = 1

def proveall(query,kb,indent=" "):
    vars = findvariables(query,[])  
    if ProveDebug > 0:
        print("%sVars in query are %s" % (indent,vars))
    answer = [['yes']]+vars+query
    if ProveDebug > 0:
        print("%sInitial answer clause is %s" % (indent,prettyclause(answer)))

    # the initial frontier is a list whose only element is the initial
    # answer clause
    frontier = [answer]
    answers = []

    while frontier:
        # give answer clause fresh variables
        answer = frontier[0]
        frontier = frontier[1:]
        if len(answer) == 1:
            yesatom = answer[0]
            answer = []
            if ProveDebug > 0:
                str = "%sFound answer:" % indent
                for var,val in zip(vars,yesatom[1:]):
                    str += " %s=%s" % (var,prettyexpr(val))
                print(str + "\n")
                for var,val in zip(vars,yesatom[1:]):
                    answer += [var,val]
                    answers.append(answer)
                    continue

        # give it fresh variables
        if ProveDebug:
            print("%sTrying to prove : %s" % (indent,prettyclause(answer)))
        answer = freshvariables(answer)
```
print("%s after fresh vars : %s" % (indent, prettyclause(answer)))

neighbors = []
firstatom = answer[1]
firstpred = firstatom[0]

if firstpred == 'not':
    result = proveall(firstatom[1:], kb, "| " + indent)
    if result == []:
        neighbor = answer[0:1] + answer[2:]
        if ProveDebug:
            print("%s Neighbor : %s" % (indent, neighbor))
        neighbors.append(neighbor)
    elif firstpred == "=":
        subs = {}
        result = unify(firstatom[1], firstatom[2], subs)
        if result != False:
            neighbor = substitute(answer[0:1] + answer[2:], subs)
            if ProveDebug:
                print("%s Neighbor : %s" % (indent, neighbor))
            neighbors.append(neighbor)
        else:
            for rule in kb:
                subs = {}
                if not unify(rule[0], answer[1], subs):
                    continue

                if ProveDebug:
                    print("%s using rule %s" % (indent, prettyclause(rule)))
                    str = ""
                    for s in subs:
                        str += " %s/%s" % (s, prettyexpr(subs[s]))
                    print("%s proven by %s with %s" %
                          (indent, prettyexpr(rule[0]), str))

                # create answer clause
                answercopy = answer[0:1] + rule[1:] + answer[2:]
                # apply substitution
                answercopy = substitute(answercopy, subs)
                if ProveDebug:
                    print("%s result: %s" % (indent, prettyclause(answercopy)))

                neighbors.append(answercopy)

    if ProveDebug:
        print("%s Found %d neighbors." % (indent, len(neighbors)))

    frontier = neighbors + frontier

if ProveDebug:
    print("%d answers: %s" % (indent, len(answers), answers))
return answers

Make sure you understand the formal that proveall is returning, which can be seen in the example included.

I am also giving you a definition of `poss` and the primitive predicates. I am also giving you what is true in the initial world. For simplicity, the knowledge base also has the goal defined.

```
kb = [[["poss", ["pickup", 'A', 'B'], 'S'],
        ['block', 'B'], ["block", 'A'],
        ['empty', 'S'], ["clear", 'A', 'S'],
```
Question 1: Queries in Situation Calculus

Before you attempt to write the forward planner, make sure you understand how to use the `proveall` procedure. Give the python code to call `proveall` to prove the following.

a is on b in the initial world.

The action of picking up a off of b is possible in the initial world.

The robot is holding a after it picks up a off of b from the initial world.

The robot’s hand is empty after putting a on the table, after picking up a off of b from the initial world.

Question 2: Multiple Answers

The previous question did not make use of `proveall`’s ability to return multiple answers. Let’s do that now. Give the python code to do the following, and report what `proveall` returns as the answer.

Find all blocks that are on the table in the initial world.

Find all actions that are possible in the initial world.

Find all actions that are possible after we have picked up a off of b from the initial world.
Find all actions that are possible after we have put a on the table after we picked up a off of b from the initial world.

**Question 3: Forward Planner**

You need to write a forward planner that explicitly keeps a frontier of worlds. The frontier will begin with just the single element `init`. On each iteration (hint: while loop), you first see if you can prove that goal is true of the current world. If it is, then you can stop. Otherwise, you need to find all actions that are possible (using `proveall`) in the current world. For each possible action, you can construct the new world as `do(Action,World)`. This will give you all of the neighbors of the current world. You then add the neighbors to the frontier using a breadth-first search strategy.

Make sure you put print statements in your code so that you can see what it is doing. Also, set ProveDebug to 0 so that you do not get the tracing messages from `proveall`. Here are the first messages from my planner.

```
World: init
Goal is not true
Neighboring worlds:
  do(pickup(a,b),init)
  do(pickup(c,t),init)

World: do(pickup(a,b),init)
Goal is not true
Neighboring worlds:
  do(putdown(a,b),do(pickup(a,b),init))
  do(putdown(a,c),do(pickup(a,b),init))
  do(putdown(a,t),do(pickup(a,b),init))

World: do(pickup(c,t),init)
Goal is not true
Neighboring worlds:
  do(putdown(c,a),do(pickup(c,t),init))
  do(putdown(c,t),do(pickup(c,t),init))

World: do(putdown(a,b),do(pickup(a,b),init))
Goal is not true
Neighboring worlds:
  do(pickup(a,b),do(putdown(a,b),do(pickup(a,b),init)))
  do(pickup(c,t),do(putdown(a,b),do(pickup(a,b),init)))

World: do(putdown(a,c),do(pickup(a,b),init))
Goal is not true
Neighboring worlds:
  do(pickup(a,c),do(putdown(a,c),do(pickup(a,b),init)))
  do(pickup(b,t),do(putdown(a,c),do(pickup(a,b),init)))

World: do(putdown(a,t),do(pickup(a,b),init))
Goal is not true
Neighboring worlds:
  do(pickup(a,t),do(putdown(a,t),do(pickup(a,b),init)))
  do(pickup(b,t),do(putdown(a,t),do(pickup(a,b),init)))
  do(pickup(c,t),do(putdown(a,t),do(pickup(a,b),init)))

World: do(putdown(c,a),do(pickup(c,t),init))
Goal is not true
Neighboring worlds:
  do(pickup(c,a),do(putdown(c,a),do(pickup(c,t),init)))
```
Hand in a copy of your code.

**Question 4**

Explain why we are not using a theorem prover that delays the **not** if it has variables in it.

**Question 5**

In the definition of the primitive relations, one must refer to the action that was executed to bring you from the previous state to the current state. In the textbook and in the slides, the definitions ensured that the action was possible in the previous state. Why is this not necessary in how you wrote your planner?

**Derived Relations in Situation Calculus**

**Question 6**

How did the Strips forward-planner that you wrote for homework 7 check if preconditions are true? Did that planner allow for the use of derived predicates? Why or why not?

**Question 7**

Describe how you would change your Strips forward-planner to allow for the use of derived predicates.

**Question 8**

Does the situation-semantics planner that you wrote for this question allow derived relations? Why or why not?

**Question 9**

Consider the block world from homework 7 where the robot can pick up and put down blocks on each other and on the table. What primitive relations could you have instead defined as derived? Give their definition. (If you test out your definitions, make sure you get rid of the corresponding primitive relations from the initial world.)
POP PLANNER

In the partial plan above, thin lines show ordering constraints and thick lines show causal links.

**Question 10**

For the partial plan above, list all threat(s) (clobberers), and explain why each is a threat, and explain how each can be resolved.

**Question 11**

For the preconditions that have not yet been satisfied, they can be satisfied using the existing actions, and without introducing more threats. Draw causal links for them.

**Prolog: Strips Forward Planner**

Review the answer to the Python Strips Forward Planner from the previous homework. We are going to make a Prolog version of this. You should also review the Prolog breadth-first theorem prover from the previous homework.

Below is the definition of the initial world, the goal, and actions, which you are required to use.

```prolog
initial([block(a), block(b), block(c), table(t), on(a,b), on(b,c), on(c,t), empty, clear(a)]).
goal([on(c,b), on(b,a), on(a,t)]).

action(pickup(A,B),
    [block(A), clear(A), block(B), on(A,B), empty],
```
[holding(A), clear(B)],
[empty, on(A, B), clear(A)]).

action(pickup(A, B),
[block(A), clear(A), table(B), on(A, B), empty],
[holding(A)],
[empty, on(A, B), clear(A)]).

action(putdown(A, B),
[holding(A), block(B), clear(B)],
[on(A, B), clear(A), empty],
[clear(B), holding(A)]).

action(putdown(A, B),
[holding(A), table(B)],
[on(A, B), clear(A), empty],
[holding(A)]).

**Question 12: Subset and Remove**

In Homework 7, I defined two procedures `subset` and `remove`. We need versions of these defined as prolog predicates for our forward planner.

Code up the `subset` predicate (including a base case) such that `subset(Sub, Set)` which expects `Sub` and `Set` to both be inputs. It should return true (succeed) if `Sub` is a subset of `Set`. You can use the member predicate.

Also, make a predicate (including a base case) for `remove(Sub, Set, Rest)` which expects `Sub` and `Set` to be inputs. It should return true if `Sub` is a setset of `Set` (in any other), and `Rest` should be the remainder of `Set` after removing `Sub`.

Other than member, do not use any other built-in predicates in your answer.

**Question 13: Neighbor**

As with the python version, we will be representing items on the frontier as pairs of `[Plan, World]`.

Define a predicate `neighbor(Top, New)` where `Top` is a list with plan and world (which is an input), and finds a neighboring pair of plan and world (`New`), that adds on an action that can be performed in the original world, and produces a new world after the actions add and delete list have been performed on it.

**Question 14: Planner**

Make a breadth first forward planner. You will define the predicate `plan(Frontier)` that if the top item on the frontier has a world that the goal is true in prints the current plan. Otherwise, it will find all neighbors of the top item, and call itself with the neighbors added to the end of the list.

Use the following code to call `plan`.

```prolog
?test() :-
    initial(Initial),
    plan([[[], Initial]]).
```