COEN 166 Artificial Intelligence

Lab Assignment #2: Vacuum Cleaner

Carlo Bilbao 1608742

Explanation of the defined functions:

goal_test: It takes an array (state) representing the current state as the input, and outputs a boolean value representing if a goal state is found. Its function is to check that the current state has both the left and right slots as clean. This designates that the current state is in a goal state, since it doesn't matter where the vacuum cleaner is currently located, just as long as both slots are clean. If both are clean, then it will return True. Otherwise, it will return false.

Actions: It takes an array (state) representing the current state as the input, and outputs a string literal that represents the most optimal next action. Its function is to determine the next action that the vacuum cleaner should take, based on the current state's information.
First, we will access the third element of the state array, which represents the current location of the vacuum (either a 0 or 1). With this value, we can use it as an index value for the array, which will correspond to the state's slot location (0 for left slot, and 1 for right slot). Overall, this represents where the vacuum cleaner is currently located. The action of 'Suck' has the highest priority. Thus, we will first check if the value in that slot is dirty, and if it is, the next action should be 'Suck.'
Else, this means that the current location is clean. Then, wherever we are, the next order of business is that we should move to the other slot. We will check if we are in the left slot first. If we are, then go to the right slot, which means the next action should be 'Right.'
The last else statement means that the vacuum is in the right slot and it is clean. So, the next action should be 'Left.'

Transition: It takes an array (state) representing the current state and a string variable (action) representing the action as the input, and then it updates the transition of the state by either changing the third element or changing the status of the current slot that the vacuum is in. Its function is to update the current state accordingly based on what the desired action is.
First, we check if the action is 'Left' by looking at the action string variable. This means that we have to change the third element of the state array (representing the location of the vacuum) to 0, which designates a left move.
Else, we check if the desired action is 'Right.' In this case, we change the third element of the state array to 1, which designates that the vacuum moved to the right slot.
The last case means that our desired action is 'Suck.' Here, we are not changing the location of the vacuum, but we are changing the cleanliness information at that current slot that the vacuum is in. We will use the third element of the state array as an index value to see where the vacuum is currently in, and then we will access that slot. Then, we will set the status of that slot to 'Clean.'

simulate: It takes an array (init_state) representing the initial state of the simulation and another array (sol) representing the solution set that stores all of the actions as the input, and prints the final solution set that was calculated along with the total cost. Its function is to simulate the entire program, containing the proper logic for the vacuum world.
We initialize the current state by taking in the first parameter init_state and then copying its information into currState. We also initialize a solution array by copying the second parameter sol. We initialize a temporary action variable which should hold information of the desired action.
Then we have a while loop, which is the meat of the simulation. The simulation should continue running until the current state has reached the goal state. Hence, we should check that the current state is not the goal state by calling the goal_test function and passing currState into it and checking that it is not true. As long as currState is not the goal state, then we should continue the simulation.
First, we call Actions, passing in currState and then set it to the temporary action variable. We are doing this to determine the next course of action. The function Actions returns the most optimal action that the vacuum should take next.
Secondly, we add this new desired action into the solution set by appending it to the solution array.
Lastly, from our current state, we transition to the next state based on the specified desired action. We will do this via the Transition function, in which we will pass in the currState and action.
We will repeat this process until the vacuum has achieved the goal state.
Finally, we print the solution set and the cost of the solution, assuming that each action has a cost of 1.

Explanation of the test case:

The test cases are handled in the main function.
We will call the simulate function and then pass in an array representing one of the 8 initial states, and then an array which is the solution set (initially empty). We do this a total of 8 times.

Test 1: ['Dirty', 'Dirty', 0]
Here, the left slot is dirty, the right slot is dirty, and the vacuum is on the left slot
We should ‘Suck,’ go ‘Right,’ then ‘Suck’

Test 2: ['Dirty', 'Dirty', 1]
Here, the left slot is dirty, the right slot is dirty, and the vacuum is on the right slot
We should ‘Suck,’ go ‘Left,’ then ‘Suck’

Test 3: ['Clean', 'Dirty', 0]
Here, the left slot is clean, the right slot is dirty, and the vacuum is on the left slot
We should go ‘Right,’ then ‘Suck’

Test 4: ['Dirty', 'Clean', 1]
Here, the left slot is dirty, the right slot is clean, and the vacuum is on the right slot

We should go ‘Left,’ then ‘Suck’

Test 5: ['Dirty', 'Clean', 0]
Here, the left slot is dirty, the right slot is clean, and the vacuum is on the left slot
We should only ‘Suck’

Test 6: ['Clean', 'Dirty', 1]
Here, the left slot is clean, the right slot is dirty, and the vacuum is on the right slot
We should only ‘Suck’

Test 7: ['Clean', 'Clean', 0]
Here, the left slot is clean, the right slot is clean, and the vacuum is on the left slot
For this initial state, we are at the goal state already; Don’t do anything

Test 8: ['Clean', 'Clean', 1]
Here, the left slot is clean, the right slot is clean, and the vacuum is on the right slot
For this initial state, we are at the goal state already; Don’t do anything

Appendix:

```
# -*- coding: utf-8 -*-
#import unittest
"""
class Testlab2(unittest.TestCase):
    def test_lab2(self):
        self.assertEqual(xxx)
"""
"""
Created on Mon Apr  4 14:51:35 2022

@author: gabby
"""


"""
Function for testing whether the state we are in is the desired state
"""
def goal_test(state):
    """
    As long as both spots are clean, it doesn't matter where we are
    """
    if state[0] == 'Clean' and state[1] == 'Clean':
        return True
    else: #One of the spots aren't clean
        return False
```

```
"""
Function for determining the next action, returns list of possible actions
"""
def Actions(state):
    #Brute force implementation, 8 states needed
    if (state[state[2]] == 'Dirty'): #If the slot we are in is dirty, suck
        return 'Suck'
    elif state[2] == 0: #If we are in the left slot and it is clean, go to the right slot
        #Go right
        return 'Right'
    else: #We are in the right slot, and it is clean
        #Go Left
        return 'Left'

"""
Function that updates state information the next action, returns state information
"""
def Transition(state, action):
    """
    Brute force implementation, 8 states needed
    """
    if action == 'Left': #If the action is left, only change the location
        state[2] = 0
    elif action == 'Right': #If the action is right, only change the location
        state[2] = 1
    else: #In this case, we are going to suck, make slot clean
        state[state[2]] = 'Clean'

def simulate(init_state, sol):
    currState = [' ',' ', 0]
    for i in range(3):
        currState[i] = init_state[i]
    print ("Initial state:", currState)
    #currState = ['Dirty', 'Clean', 1]
    solution = sol
    #cost = 0

    #possActions = Actions(currState) #A list of possible actions in the current state
    action = [' '] #The current action
    #print (Actions(currState))
    #print (possActions)
    #print (possActions[1])
```

```
    #print (goal_test(currState))

    #possActionsIndex = 0
    """
    Handles the simulation, ends when the current state is the goal state
    """
    while goal_test(currState) != True:
        action = Actions(currState) #Determine the next course of action
        solution.append(action) #Add this action to the solution set
        #temp = Transition(currState, action) #Update the current state to the next state
        #for i in range(3):
            #currState[i] = temp[i] #Go to new state
        Transition(currState, action)

    print('Final Solution:', solution, ', Cost:', len(solution))

"""
Main function, executes code
"""
if __name__=='__main__':
    #unittest.main()

    print ("Test 1 :")
    simulate(['Dirty', 'Dirty', 0], []) #Left dirty, Right dirty, Location left
    print ("")

    print ("Test 2 :")
    simulate(['Dirty', 'Dirty', 1], []) #Left dirty, Right dirty, Location left
    print ("")

    print ("Test 3 :")
    simulate(['Clean', 'Dirty', 0], []) #Left clean, Right dirty, Location left
    print ("")

    print ("Test 4 :")
    simulate(['Dirty', 'Clean', 1], []) #Left dirty, Right clean, Location right
    print ("")

    print ("Test 5 :")
    simulate(['Dirty', 'Clean', 0], []) #Left dirty, Right clean, Location left
    print ("")

    print ("Test 6 :")
    simulate(['Clean', 'Dirty', 1], []) #Left dirty, Right clean, Location right
```

```
print ("")

print ("Test 7 :")
simulate(['Clean', 'Clean', 0], []) #Left clean, Right clean, Location left
print ("")

print ("Test 8 :")
simulate(['Clean', 'Clean', 1], []) #Left clean, Right clean, Location right
print ("")
```