DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

AHMADU BELLO UNIVERSITY, ZARIA.

INTELLIGENT AGENTS

AMINU, USMAN KABIR

15th April, 2017.

QUESTION

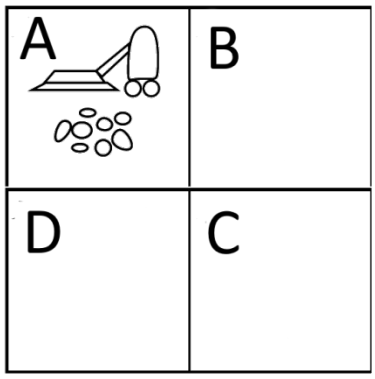
Implement a table-lookup agent for the special case of the vacuum-cleaner world consisting of a 2 2 grid of open squares, in which at most two squares will contain dirt. The agent starts in the upper left corner, facing to the right. Recall that a table-lookup agent consists of a table of actions indexed by a percept sequence. In this environment, the agent can always complete its task in nine or fewer actions (four moves, three turns, and two suck-ups), so the table only needs entries for percept sequences up to length nine. At each turn, there are eight possible percept vectors, so the table will be of size $\sum_{i=1}^{9} 8^i = 153,391,688$

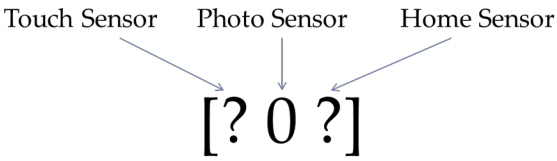
Fortunately, we can cut this down by realizing that the touch sensor and home sensor inputs are not needed; we can arrange so that the agent never bumps into a wall and knows when it has returned home. Then there are only two relevant percept vectors, ?0? and ?1?, and the size of the table is at most $\sum_{i=1}^{9} 2^i = 1022$

Run the environment simulator on the table-lookup agent in all possible worlds (how many are there?). Record its performance score for each world and its overall average score.

VACUUM CLEANER

In the Simple World, We are given a vacuum cleaner, and tasked to clean up a 2 x 2 open square, namely A, B, C and D as shown below:





The Agent is Characterized with the following properties:

1. Can Move Right, Left, Up & Down
2. The agent starts in the upper left corner, facing to the right
3. The agent has a three-element percept vector on each turn. The first element, a touch sensor, should be a 1 if the machine has bumped into something and a 0 otherwise. The second a photo-sensor under the machine, which emits a 1 if there is dirt there and a 0 otherwise. The third comes from an infrared sensor, which emits a 1 when the agent is in its home location, and a 0 otherwise.

The Environment is characterized with the following properties:

1. It is an open space (no obstacles).

It is therefore explicitly clear from the question that:

The touch sensor and home sensor inputs are not needed. Hence, our percept vectors are:

[?0?] or [?1?]

This implies that only the Photo Sensor will be used in this question.

POSSIBLE WORLDS

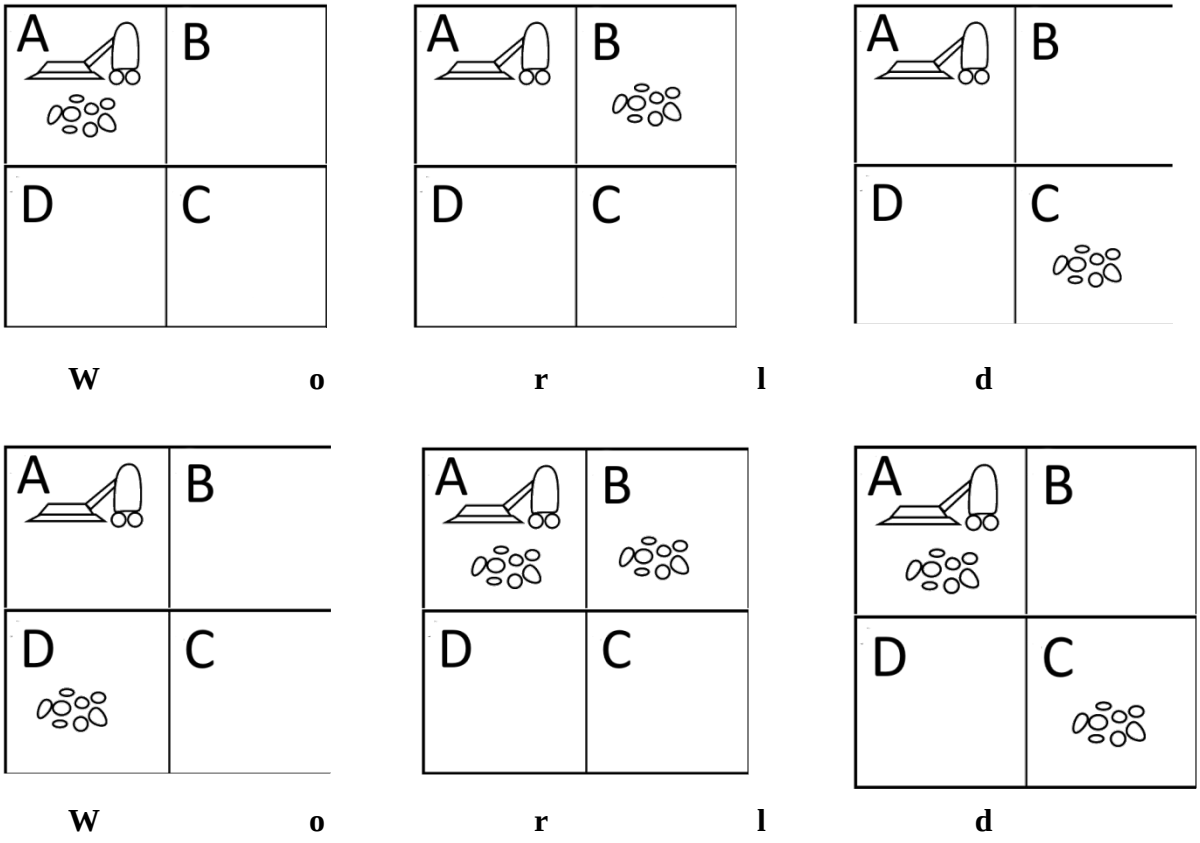
The Number of possible worlds will be the certain worlds where the photo sensor senses dirt. It is stated from the question that there can be dirt in **Atmost 2** squares.

Which can therefore be of either None, A,B,C,D,AB,AC,AD,BC,BD,CD=11 Possible worlds.

Mathematically obvious,

$$^4C_2 + ^4C_1 + ^4C_0 = 11 \text{ Possible worlds.}$$

The Vacuum cleaner is already initialized to be in the upper left corner, facing to the right (though is not our concern, giving that we need only two percept vectors).



MOVES TO GOAL

From an initialize state, CELL A, We can move to each of the respective cells where to perform the action.

The table below shows the sequence of actions required to achieve the goal for each world.

WORLD	1st Move (CELL A)	2nd Move (CELL B)	3rd Move (CELL C)	4th Move (CELL D)
A	?1?			
B	?0?	?1?		
C	?0?	?0?	?1?	
D	?0?	?0?	?0?	?1?
AB	?1?	?1?		
AC	?1?	?0?	?1?	
AD	?1?	?0?	?0?	?1?
BC	?0?	?1?	?1?	?0?
BD	?0?	?1?	?0?	?1?
CD	?0?	?0?	?1?	?1?
Noop	?0?	?0?	?0?	?0?

PERFORMANCE MEASURES

To obtain the Performance Scores, we use the following measures and their declarations:

a. **Score**

   i. Suck = 100 points

   ii. Action (Move or Rotate)= 1 point

b. **Amount Of Dirts Collected**

   i. Suck = 1

   ii. Action (Move or Rotate)= 0

c. **Amount Of Time**

   i. Suck = 5 seconds

   ii. Action= 2 seconds

d. **Amount Of Electricity Used**

   i. Suck = 15 Units (N/kwh)

   ii. Action (Move or Rotate) = 2 Units (N/Kwh)

PERFORMANCE SCORE

WORLD	Score	Dirt	Time	Units
A	**100**	**1**	**5**	**15**
B	**101**	**1**	**7**	**17**
C	**102**	**1**	**9**	**19**
D	**103**	**1**	**11**	**21**
AB	**201**	**2**	**12**	**32**
AC	**202**	**2**	**14**	**34**
AD	**203**	**2**	**16**	**36**
BC	**203**	**2**	**16**	**36**
BD	**203**	**2**	**16**	**36**
CD	**203**	**2**	**16**	**36**
Noop	**3**	**0**	**6**	**6**
AVERAGE	**147.6364**	**1.454545**	**11.63636**	**26.18182**

CONCLUSION

- The Average Score is 147.64
- Average Dirt is 1.45 Dirt
- Average Time is 11.63 Seconds
- Average Electricity Consumed is 26.18 Kwh

After running the above programs across various possible worlds, the implication of the above measures is that, The agent or product can be designed based on the above specifications.

MAIN PROGRAM

```
from environment import VacuumEnvironment
from agents import RandomAgent, ReflexAgent, InternalAgent

ENV_SIZE = (12, 12)
# change this variable to the environment size prof gave in his question

DIRT_CHANCE = 0.05
# this holds the possibility of encountering a dirty tile

def main():
    env = VacuumEnvironment(ENV_SIZE, DIRT_CHANCE)
    agent = InternalAgent()
"""
while creating the code i had no means of creating a switch like statment
structure in python so
manually change the agent type here based on the type of agent you are
testing , the agents.py file contains the
functions
"""

    print env.room[0]
    print env.room[1]

    observation = env.state()
    reward = 0
    done = False
    action = agent.act(observation, reward)
    turn = 1

    while not done:

        observation, reward, done = env.step(action[0])
        print "Step {0}: Action - {1}".format(turn, action[1])

        action = agent.act(observation, reward)
        # print "Reward {0}   Total Reward {1}".format(reward, agent.reward)
        turn += 1

    print env.room[0]

if __name__ == "__main__":
    main()
```

AGENTS

```
 import numpy as np
import string

"""the actions are coded in an array which is referenced based on rules that
are satisfied
or read from our percepts.
this structure cann also serve as a lookup table of sorts, never got round to
implementing the lookup
agent due to the confusing nature of the second question"""

ACTIONS = ((0, "Go Forward"),  # actions were done based on directions facing
from 1 to 4
           (1, "Turn Right"),
           (2, "Turn Left"),
           (3, "Suck Dirt"),
           (4, "Turn Off"),
           (-1, "Break"),)

Lookup = {'Go Forward': 0, 'Turn Right': 1, 'Turn Left': 2, 'Suck Dirt': 3,
'Turn Off': 4, 'Break': -1}
# sample lookup table as a dictionary

""" this here is an agent that randomises its actions it does not really care
much about the state of the environment
i coded it to just pick random actions from my actions array"""


class RandomAgent(object):
    def __init__(self):
        self.reward = 0

    def act(self, observation, reward):
        self.reward += reward

        action = ACTIONS[np.random.randint(len(ACTIONS))]
        return action

""" incomplete  question 2 ., lookuptable needed before i can proceed , if
you guys can provide the lookup table
 in any of the formats i gave in the beginning of this file i.e ACTIONS,
Lookup"""
class TableLookupAgent(object):
    def __init__(self):
        self.reward = 0

    def act(self, observation, reward):
        self.reward += reward

        action = Lookup[observation]
        return action
```

```
""" so i got stuck thinking of how to  create the lookup table based agent in
profs assignment, it should normally
work like this reflex agent, now the remaining group members should find a
way to create a lookup table for the actions
 and a way to search through the table based on percept input i.e change the
rule based actions i hard coded below"""


class ReflexAgent(object):
    def __init__(self):
        self.reward = 0

    def act(self, observation, reward):
        self.reward += reward

        # If dirt then suck
        if observation['dirt'] == 1:
            return ACTIONS[3]

        # If obstacle then turn
        if observation['obstacle'] == 1:
            return ACTIONS[1]

          # Else randomly choose from first 3 actions (stops infinite loop
circling edge)
        return ACTIONS[np.random.randint(3)]


""" this internal agent class was supposed to be a self updating agent wanted
to implement the utility or goal agent
type , if you are brave enough you can go through it and fix it , as far as i
know it works and no nasty comments
about going hyper
"""


class InternalAgent(object):
    def __init__(self):
        self.reward = 0
         self.map = [[-1, -1], [-1, -1]]  # 0-Empty, 1-Dirt, 2-Obstacle, 3-
Home

        # Agent's relative position to map and direction
        self.x = 0
        self.y = 0
        self.facing = 0  # -1-Unknown, 0-Up, 1-Right, 2-Down, 3-Left

    def add_map(self):

        side = self.is_on_edge()

        while side >= 0:
```

```
            if side == 0:  # Top
                self.map.insert(0, [-1] * len(self.map[0]))
                self.x += 1

            elif side == 1:  # Right
                for row in self.map:
                    row.append(-1)

            elif side == 2:  # Down
                self.map.append([-1] * len(self.map[0]))

            elif side == 3:  # Left
                for row in self.map:
                    row.insert(0, -1)
                self.y += 1

            side = self.is_on_edge()

    def is_on_edge(self):
        if self.x == 0:
            return 0

        elif self.y == len(self.map[0]) - 1:
            return 1

        elif self.x == len(self.map) - 1:
            return 2

        elif self.y == 0:
            return 3

        return -1

    def move_forward(self):
        if self.facing == 0:
            self.x -= 1

        elif self.facing == 1:
            self.y += 1

        elif self.facing == 2:
            self.x += 1

        elif self.facing == 3:
            self.y -= 1

    # If obstacle in position then move back to previous square
    def move_backwards(self):
        if self.facing == 0:
            self.x += 1

        elif self.facing == 1:
            self.y -= 1
```

```
        elif self.facing == 2:
            self.x -= 1

        elif self.facing == 3:
            self.y += 1

    def update_map(self, observation):
        if observation['dirt'] == 1:
            self.map[self.x][self.y] = 1

        elif observation['home'] == 1:
            self.map[self.x][self.y] = 3

        else:
            self.map[self.x][self.y] = 0

        if observation['obstacle'] == 1:
            self.map[self.x][self.y] = 2
            self.move_backwards()

        # Fill in borders
        x_len = len(self.map) - 1
        y_len = len(self.map[0]) - 1

        if self.map[0][1] == 2 and self.map[1][0] == 2:
            self.map[0][0] = 2

        if self.map[0][y_len - 1] == 2 and self.map[1][y_len] == 2:
            self.map[0][y_len] = 2

        if self.map[x_len - 1][0] == 2 and self.map[x_len][1] == 2:
            self.map[x_len][0] = 2

         if self.map[x_len][y_len - 1] == 2 and self.map[x_len - 1][y_len] ==
2:
            self.map[x_len][y_len] = 2

     # Determine next action needed to move towards next_square from current
position
    def next_step(self, next_square):
        if next_square[0] < self.x and self.facing != 0 and self.map[self.x -
1][self.y] != 2:
            action = ACTIONS[2]

        elif next_square[0] < self.x and self.facing == 0 and self.map[self.x
- 1][self.y] != 2:
            action = ACTIONS[0]

        elif next_square[0] > self.x and self.facing != 2 and self.map[self.x
+ 1][self.y] != 2:
            action = ACTIONS[2]

        elif next_square[0] > self.x and self.facing == 2 and self.map[self.x
+ 1][self.y] != 2:
```

```
            action = ACTIONS[0]

                elif  next_square[1]  >  self.y  and  self.facing  !=  1  and
self.map[self.x][self.y + 1] != 2:
            action = ACTIONS[2]

                elif  next_square[1]  >  self.y  and  self.facing  ==  1  and
self.map[self.x][self.y + 1] != 2:
            action = ACTIONS[0]

                elif  next_square[1]  <  self.y  and  self.facing  !=  3  and
self.map[self.x][self.y - 1] != 2:
            action = ACTIONS[2]

                elif  next_square[1]  <  self.y  and  self.facing  ==  3  and
self.map[self.x][self.y - 1] != 2:
            action = ACTIONS[0]

        else:
            action = ACTIONS[4]

        # If moving forward check if map needs to be expanded
        if action[0] == 0:
            self.move_forward()

        if action[0] == 2:
            self.facing = (self.facing - 1) % 4

        return action

    def find_nearest(self, square_type):
        # Else move towards nearest unknown
        min_dist = None
        next_square = None

        for i, row in enumerate(self.map):
            for j, square in enumerate(row):
                if square == square_type:
                    dist = (self.x - i) ** 2 + (self.y - j) ** 2
                    if min_dist is None or dist < min_dist:
                        min_dist = dist
                        next_square = (i, j)

        return next_square

    def choose_action(self):
        # If on a patch of dirt then suck it up
        if self.map[self.x][self.y] == 1:
            return ACTIONS[3]

        next_square = self.find_nearest(-1)

        # If no more unknowns then head home
        if next_square is None:
```

```
            next_square = self.find_nearest(3)

        return self.next_step(next_square)

    def act(self, observation, reward):
        self.reward += reward

        self.update_map(observation)
        self.add_map()

        # Choose action (based on map)
        return self.choose_action()
```

ENVIRONMENTS

```
        """
Vacuum Cleaner Environment
"""
import numpy as np


class VacuumEnvironment(object):
    def __init__(self, size, dirt):
        self.size = size
        self.dirt = dirt

        self.agent_x = np.random.randint(self.size[0])
        self.agent_y = np.random.randint(self.size[1])
         self.agent_facing = np.random.randint(4)    # 0-up, 1-right, 2-down,
3-left

        # Layer 0: dirt, layer 1: objects/home
        self.room = np.zeros((2, self.size[0], self.size[1]))

        for row in range(self.size[0]):
            for col in range(self.size[1]):
                if np.random.uniform() < self.dirt:
                    self.room[0][row][col] = 1

        # Set home base
        home_x = np.random.randint(self.size[0])
        home_y = np.random.randint(self.size[1])
        self.room[1][home_x][home_y] = 1

    def state(self, obstacle=False):
        return {"obstacle": int(obstacle),
                "dirt": self.room[0][self.agent_x][self.agent_y],
                "home": self.room[1][self.agent_x][self.agent_y],
```

```
                "agent": (self.agent_x, self.agent_y)}

    def has_hit_obstacle(self):
        if (self.agent_facing == 0 and self.agent_x == 0) or \
           (self.agent_facing == 1 and self.agent_y == self.size[1] - 1) or \
           (self.agent_facing == 2 and self.agent_x == self.size[0] - 1) or \
           (self.agent_facing == 3 and self.agent_y == 0):
            return True

        return False

    def move_forward(self):
        """
        Updates agents position
        :return: Whether agent hit obstacle
        """
        if self.has_hit_obstacle():
            return True

        if self.agent_facing == 0:
            self.agent_x -= 1

        elif self.agent_facing == 1:
            self.agent_y += 1

        elif self.agent_facing == 2:
            self.agent_x += 1

        elif self.agent_facing == 3:
            self.agent_y -= 1

        return False

    def step(self, action):
        obstacle = False
        reward = -1  # Default -1 for each action taken
        done = False

        if action == 0:
            obstacle = self.move_forward()

        elif action == 1:
            self.agent_facing = (self.agent_facing + 1) % 4

        elif action == 2:
            self.agent_facing = (self.agent_facing - 1) % 4

        elif action == 3:
            # Reward of +100 for sucking up dirt
            if self.room[0][self.agent_x][self.agent_y] == 1:
                reward += 100
                self.room[0][self.agent_x][self.agent_y] = 0

        elif action == 4:
```

```
            # If not on home base when switching off give reward of -1000
            if self.room[1][self.agent_x][self.agent_y] != 1:
                reward -= 1000

            done = True

        return self.state(obstacle), reward, done
```