Peter Subacz – CS534: Artificial Intelligence – 9/15/19

Contents

Chapter 2 – Ex 2.3

A. False – An agents goals can still be achievable with partially observable information, see page 42.
B. True – See page 49, if the task environment is unobservable than no pure reflex agent can be rational. Reflex agents respond to percepts, if a percept cannot be observed or interpreted than the agent fails to respond.
C. True – Given all possibly task environments, with an infinite number of task environments, the law of total probability that there will be a case for an agent to be rational in that environments.
D. False – See page 46, the agent program take the current percept as an input. The agent functions with the entire percept history.
E. True – Depends on the tasking and abstractions/assumptions made. The agent function maps a given percept to an action
F. True – An agent randomly selecting actions can be rational in a deterministic task environment. The rationality of this agent would depend on its performance measure and the agent function over the run time of the agent.
G. True – An agent designed to play Sudoku on a MxN board would still function if the board scales in size (expanded Sudoku is not limited to integers form 1-9)
H. False – Not every agent is ration in an unobservable environment. An agent may still be able to accomplish its goals.
I. False – A perfectly rational agent able to play cards such as poker or blackjack may still lose a game due to random factors (such as luck).

Chapter 2 – Ex 2.9

Python code appended to end of document.

Chapter 3 – Ex 3.2

As Defined on page 67, the initial state, actions, and transition model implicitly define the space state.

A. Maze/Robot Configuration 1 - Your goal is to navigate a robot out of a maze. The robot starts in the center of the maze facing north. You can turn the robot to face north, east, south, or west. You can direct the robot to move forward a certain distance, although it will stop before hitting a wall.

   - States: The position and orientation of the robot.
   - Initial State: Center of the maze with a northward orientation.
   - Actions: Orient robot a new direction (4 possible actions), Advance position (1 action).
   - Transition Model: Orient robot turns a robot one of four possible directions and move forward advances the position of the robot. Returns a new position orientation of the robot every step.
   - Goal Test: Check to see if robot has left the maze.
   - Path cost: None or one per action depends on how problem is implemented.

   In the graph of possible states, a agent can take 5 possible actions with 1 action being redundant (loopy path, a robot can orientate a to the same direction). This without pruning looping paths, the space state is infinitely large. If the graph is pruned of loopy paths, each action only generates 4 possible nodes in a graph (3 turn actions and 1 advance action). Therefore 4n possible subset of actions is generated each step. If we consider unique paths to be P_i which covers all paths from node n_0 to n_i then the space state will be the $\sum P_i$.

B. Maze/Robot Configuration 2 - Your goal is to navigate a robot out of a maze. The robot starts in the center of the maze facing north. You can turn the robot to face north, east, south, or west. You can direct the robot to move forward a certain distance, although it will stop before hitting a wall.

   - States: The position and orientation of the robot.
   - Initial State: Center of the maze with a northward orientation
   - Actions: Advance position, If intersection $\geq$ 2 corridors then the robot can turn to a new orientation.
   - Transition Model: The robot advances in an orientation until a wall has been detected. If a wall or intersection exists, a turn action becomes available.
   - Goal Test: Check to see if robot has left the maze
   - Path cost: None or one per action, depends on how problem is implemented

   Similarly to A, the space state is infinite due to loopy paths that can be generated. If the loopy paths are pruned, the unique paths to be P_i which covers all paths from node n_0 to n_i then the space state will be the $\sum P_i$ is much smaller than the $\sum P_i$ in A.

C. Maze/Robot Configuration 3 - From each point in the maze, we can move in any of the four directions until we reach a turning point, and this is the only action we need to do.

Reformulate the problem using these actions. Do we need to keep track of the robot's orientation now?-

- States: The position of the robot.
- Initial State: Center of the maze
- Actions: Advance 1 of 4 directions
- Transition Model: Move a direction until an intersection has been encountered.
- Goal Test: Check to see if robot has left the maze
- Path cost: None or one per action, depends on how problem is implemented

Similar to A and B, the space state is infinite with loopy paths. However, the unique paths to be P_i which covers all paths from node n_0 to n_i is significant smaller than the $\sum P_i$ of A and $\sum P_i$ of B. This is due to turn actions being eliminated and single advanced actions being removed. The robot orientation is no longer needed due to advanced actions allowing movement in 4 possible directions.

D. Abstractions made

- The maze only allows for 4 possible move actions.
- The robots sensors need to determine location and orientation.
- Physically moving the robot from point A to point B.
- The maze only has straight walls and the only allow for linear travel.

Chapter 2 – Ex 3.3

E. Suppose friend A and B live in two arbitrary cities within the Romanian map in Figure 2.8. The problem can be solved with a bi-directional graph search with a heuristic.

- States: A graph which contains the location of a map [m(x,y)] and the defined paths [p(x,y)] between each city on the map.
- Initial State: A graph starting at the location of A (x,y) and a graph starting at the location of B(x,y). Each the path of A is contained within $P_A(x,y)$ and the path of B is contained within $P_B(x,y)$.
- Actions: A friend moves to a neighboring city from a previous city.
- Transition Model: Move each person to a city and wait for both to arrive at each city. The transition model returns a new location for A (x,y) and B (x,y).
- Goal Test: Both friends are in each city such that A(x,y) = B(x,y).
- Path cost: the longest time required fro either friend to traverse to a new city such that argmax ($P_A(x,y)$,$P_B(x,y)$).

F. As stated on page 94, the admissibly of a heuristic can never overestimate the cost to reach the goal

i) D(i,j) is admissible, Take the path from Bucharest to Craiova as an example. The total cost to is 101 +138 = 239. The heuristic being used here is a straight line from Bucharest to Craiova which can be calculated via Pythagoras theorem to be sqrt(a^2+b^2) to be 171.01 (An abstraction being made here is that this map is using Euclidean geometry...). Extending this to other cities will still yield an admissible heuristic because the shortest distance between two points is a straight line. A different method of proving admissibility would be by the triangle inequality theorem.

ii) 2*D(i,j) is inadmissible due to the ability to overestimate the path cost. Using the previous example, this heuristic would yield a distance of 342.02.

iii) D(i,j)/2 is admissible because it is much smaller than a heuristic generated by D(i,j)

G. A completed connected map exists where no solution exists, for example: a graph of two connected nodes. Since the problem states that each friend must move, the friends will never be in the city at the same time. If loopy paths are allowed, then one friend can stand still and the other can travel to the new city.

H. Yes, as per the problem rules, some friends will need to travel to the same city twice. The problem states that each friend must travel a distance to a city. In order to avoid the problem presented in C, there can be a map where a friend will visit the same city twice. Take a Romanian map with a friend living on Vaslui and Pitesti. These friends will never meet because they will pass each other when traversing form/to Burcharest and Urziceni. This requires a circular path to make sure they meet.

a) Friend 1's path to travel: Vaslui→Urziceni→Bucharest→Pitesti

b) Friend 2 's path to travel: Pitesti→Craiova→Rimnicu Vilcea→Pitesti

Python Code

```
# -*- coding: utf-8 -*-
"""
Created on Sun Sep 15 11:11:16 2019
@author: psubacz
As stated on page 38:
Consider the simple vacuum-cleaner agent that cleans a square if it is dirty and moves to the
other square if not; this is the agent function tabulated in Figure 2.3. Is this a rational agent?
That depends! First, we need to say what the performance measure is, what is known about
the environment, and what sensors and actuators the agent has. Let us assume the following:

• The performance measure awards one point for each clean square at each time step,
over a “lifetime” of 1000 time steps.

• The “geography” of the environment is known a priori (Figure 2.2) but the dirt distribution
and the initial location of the agent are not. Clean squares stay clean and sucking
cleans the current square. The Left and Right actions move the agent left and right
except when this would take the agent outside the environment, in which case the agent
remains where it is.

• The only available actions are Left , Right, and Suck.

• The agent correctly perceives its location and whether that location contains dirt.

• The dirt distrubtion has been abstracted in this enviroment to be a square that is dirty or clean
```

```
NOTE: There is a slight variance with the performance measure due to the random generation of which
   floor the robot spawn in! Therefore if the robot spawns on a clean floor then moves to a dirty floor
   then the performance measure should read 999. If the robot spawn on a dirty floor the performance
   measure should read 1000
"""
import numpy as np
import random


class Floor:
   '''
   Class to contain floor object with attributes
   '''
   def __init__(self,location,isDirty = True):
      '''
      Initialize Floor class for each instance of floor object
      '''
      self.location = location
      self.isDirty = isDirty
      self.dirtDistribution = None


   def get_location(self):
      '''
      Returns name of floor space
      '''
      return self.location
```

```
    def is_dirty(self):
        '''
        Returns True if floor is dirty and False is floor is clean.
        '''
        return self.isDirty

    def set_dirty(self):
        '''
        Sets isDirty attribute to True, indacates floor has been made dirty
        '''
        self.isDirty = True

    def set_clean(self):
        '''
        Sets isDirty attribute to False, indacates floor has been made clean
        '''
        self.isDirty = False

class Robot:
    def __init__(self, position = None, debug = False,preset_clean_floor = None):
        '''
        Initialize robot with random position if None is given, and Floor ennviroment
        '''
        #Assumption is that the enviroment is a known priori  but the dirt distribution
```

```
        #and the initial location of the agent are not.
        self.position = position
        self.debug = debug
        #Performance Measures are incremented when a floor has been detected as clean
        self.performance_measure = 0
        self.performed_actions = 0
        self.percept_sequence = []
        #There are two known floor Locations, left = 0 and right = 1
#       if (preset_clean_floor == None):
#           #Both floors are dirty
#           self.floor_locations = [Floor('left'),Floor('right')]
        if(preset_clean_floor==0):
            self.floor_locations = [Floor('left',False),Floor('right')]
        elif(preset_clean_floor==1):
            self.floor_locations = [Floor('left'),Floor('right',False)]
        elif(preset_clean_floor==2):
            self.floor_locations = [Floor('left',False),Floor('right',False)]
        else:
            self.floor_locations = [Floor('left'),Floor('right')]
        #Generate floor Location
        self.get_robot_position()

    def get_robot_position(self):
        '''
        '''
```

```
    #If robot position is unknown, randomyl generate a location from possible floor locations
    if (self.position == None):
        self.position = random.randint(0, len(self.floor_locations)-1)
    return self.position

def move_left(self):
    '''
    '''
    #The Left actions move the agent left except when this would take the agent
    #outside the environment, in which case the agent remains where it is.
    self.position = 0

def move_right(self):
    '''
    '''
    #The Right actions move the agent right except when this would take the agent
    #outside the environment, in which case the agent remains where it is.
    self.position = 1

def clean_floor(self,floor_location):
    '''
    '''
    #Set floor to clean(false) and
    while(self.percieve_dirt()):
        self.floor_locations[self.position].set_clean()
```

```
def percieve_dirt(self):
    '''
    Percieve if the floor is dirty, return true if dirty and false if clean
    '''
    if (self.floor_locations[self.position].is_dirty()==True):
        return True
    else:
        return False


def reflex_agent(self):
    '''
    A Simple Reflex agent as stated on page 49:
    function SIMPLE-REFLEX-AGENT(percept ) returns an action
        persistent: rules, a set of condition–action rules
        state←INTERPRET-INPUT(percept )
        rule←RULE-MATCH(state, rules)
        action ←rule.ACTION
        return action
    '''
    #Percepts
    state = self.percieve_dirt()
    just_cleaned = False

    #Log the percept sequence
```

```
        self.percept_sequence.append((self.position,state))

        #Rule 1 - Clean a dirty floor
        if state:
            #Action
            self.clean_floor(self.floor_locations[self.position])
            self.performed_actions += 1
#           print('Floor Cleaned')
            self.performance_measure += 1
            just_cleaned =True

        #Rule 2 -  Floor clean, Move to next area
        elif (self.position == 1):
            #Action
            self.move_left()
            self.performed_actions += 1
#           print('Moving left...')

        #Rule 3 -  Floor clean, Move to next area
        elif (self.position == 0):
            #Action
            self.move_right()
            self.performed_actions += 1
#           print('Move right...')
```

```
        #Reward - Reward 1 point if floor is clean dont double count points!
        if (self.floor_locations[self.position].is_dirty() == False) and (just_cleaned == False):
            self.performance_measure += 1

        #Debug print statements
        if (self.debug == True):
            print("self.performance_measure: ",self.performance_measure)
            print("self.performed_actions: ",self.performed_actions)

    def get_stats(self):
        '''
        Returns the a tuple of actions, performance measure, percept sequence
        '''
        return (self.performed_actions,self.performance_measure,self.percept_sequence)


def run_enviroment(MAX_TIME_STEPS = 1000):
    '''
    '''
    #Robot - Both floors dirty
    cleaningRobot = Robot()
    #Run Enviroment for MAX_TIME_STEPS
    for step in range(0,MAX_TIME_STEPS):
        cleaningRobot.reflex_agent()
    stats0 = cleaningRobot.get_stats()
```

```
print('\nRobot - Both floors dirty')
print('performed_actions: ',stats0[0])
print('performance_measure: ',stats0[1])
print('Avg performance_measure: ',stats0[1])
#print('percept_sequence: ', stats0[2])
#clean up
del cleaningRobot

#Robot - left floor dirty
cleaningRobot = Robot(preset_clean_floor = 0)
#Run Enviroment for MAX_TIME_STEPS
for step in range(0,MAX_TIME_STEPS):
    cleaningRobot.reflex_agent()
stats1 = cleaningRobot.get_stats()
print('\nRobot - left floor dirty')
print('performed_actions: ',stats1[0])
print('performance_measure: ',stats1[1])
#print('percept_sequence: ', stats1[2])
#clean up
del cleaningRobot

#Robot - right floor dirty
cleaningRobot = Robot(preset_clean_floor = 1)
#Run Enviroment for MAX_TIME_STEPS
for step in range(0,MAX_TIME_STEPS):
```

```
        cleaningRobot.reflex_agent()
    stats2 = cleaningRobot.get_stats()
    print('\nRobot - right floor dirty')
    print('performed_actions: ',stats2[0])
    print('performance_measure: ',stats2[1])
    #print('percept_sequence: ', stats2[2])
    #clean up
    del cleaningRobot

    #Robot - Both floors clean
    cleaningRobot = Robot(preset_clean_floor = 2)
    #Run Enviroment for MAX_TIME_STEPS
    for step in range(0,MAX_TIME_STEPS):
        cleaningRobot.reflex_agent()
    stats3 = cleaningRobot.get_stats()
    print('\nRobot - Both floors clean')
    print('performed_actions: ',stats3[0])
    print('performance_measure: ',stats3[1])
    #print('percept_sequence: ', stats3[2])
    #clean up
    del cleaningRobot
    APM = (stats0[1]+stats1[1]+stats2[1]+stats3[1])/4
    print('\nAvg Performance Measure: ', APM)

if __name__ == "__main__":
```

```
'''
Run enviroment and display performance_measure
'''
run_enviroment()
```