CS 543: Homework Assignment 1

Aditya Mehrotra
amehrotra@wpi.edu

1 Chp 2, Ex 2.3

a. An agent that senses only partial information about the state cannot be perfectly rational.

False. The rationality of an agent depends on its decisions based on the information available to it.

b. There exist task environments in which no pure reflex agent can behave rationally.

True. Pure reflex agents do not keep a track of past percepts and thus can't make correct decisions in a partially observable environment.

c. There exists a task environment in which every agent is rational.

True. In an environment with only a single state where any action would have the same consequence.

d. The input to an agent program is the same as the input to the agent function.

False. Input to an agent program is the current input perceived by the agent's sensors whereas to the agent function is the entire percept sequence.

e. Every agent function is implementable by some program/machine combination.

False. Since there can exist an agent without a sensor, the agent function would not have any percept to map to an action. Thus, there are some agent functions that cannot be implemented using a program/machine combination.

f. Suppose an agent selects its action uniformly at random from the set of possible actions. There exists a deterministic task environment in which this agent is rational.

True. The agent would be considered rational where the task is to be unpredictable.

g. It is possible for a given agent to be perfectly rational in two distinct task environments.

True. Suppose an agent's performance is based on two independent tasks, placing it in an environment which where one of the tasks is rewarded can still help it be rational.

h. Every agent is rational in an unobservable environment.

False. The agent may perform certain tasks that have high risk rather than reward in an unobservable environment.

i. A perfectly rational poker-playing agent never loses.

False. Unless the agent in question is an omniscient agent, it will most likely lose against an unobservalbe, stochastic environment.

2 Chp 2, Ex 2.9

Here, we assume that each action (Suck, Left, Right) costs 1 performance measure.

```
import random


class Environment(object):
    def __init__(self):
        self.status = {'A': random.randint(0, 1), 'B': random.randint(0, 1)}


class SRAgent(Environment):
    Performance = 0

    def __init__(self, Environment):
        self.status = Environment.status
        self.sequence()

    def SimpleReflexAgent(self, status, location):
        if status == 'Dirty':
            self.Performance += 1
            return 'Suck'
        elif location == 'A':
            self.Performance += 1
            return 'Right'
        elif location == 'B':
            self.Performance += 1
            return 'Left'

    def sequence(self):
```

```
vacuumLocation = random.randint(0, 1)

if vacuumLocation == 0:
    print("Vacuum is at location A.")

    if self.status['A'] == 1:
        print("Location A is dirty.", self.SimpleReflexAgent(
            'Dirty', 'A'))
        self.status['A'] = 0
        print("Location A has been cleaned.")

        if self.status['B'] == 1:
            print("Moving to location B: ", self.SimpleReflexAgent(
                'Clean', 'A'))

            print("Location B is dirty.", self.SimpleReflexAgent(
                'Dirty', 'B'))
            self.status['B'] = 0
            print("Location B has been cleaned.")

    else:
        if self.status['B'] == 1:
            print("Moving to location B: ", self.SimpleReflexAgent(
                'Clean', 'A'))

            print("Location B is dirty.", self.SimpleReflexAgent(
                'Dirty', 'B'))
            self.status['B'] = 0
            print("Location B has been cleaned.")

elif vacuumLocation == 1:
    print("Vacuum is at location B.")

    if self.status['B'] == 1:
        print("Location B is dirty.", self.SimpleReflexAgent(
            'Dirty', 'B'))
        self.status['B'] = 0
        print("Location B has been cleaned.")
```

```
            if self.status['A'] == 1:
                print("Moving to location A: ", self.SimpleReflexAgent(
                    'Clean', 'B'))
                print("Location A is dirty.", self.SimpleReflexAgent(
                    'Dirty', 'A'))
                self.status['A'] = 0
                print("Location A has been cleaned.")

        else:
            if self.status['A'] == 1:
                print("Moving to location A: ", self.SimpleReflexAgent(
                    'Clean', 'B'))

                print("Location A is dirty.", self.SimpleReflexAgent(
                    'Dirty', 'A'))
                self.status['A'] = 0
                print("Location A has been cleaned.")


environment = Environment()
print('Environment status before running agent: ', environment.status)
vacuum = SRAgent(environment)
vacuum.status = environment.status
print('Environment status after running agent: ', vacuum.status)
print('Performance Score: ', vacuum.Performance)
```

Typical output:

```
Environment status before running agent:  {'A': 1, 'B': 0}
Vacuum is at location B.
Moving to location A:  Left
Location A is dirty. Suck
Location A has been cleaned.
Environment status after running agent:  {'A': 0, 'B': 0}
Performance Score:  2
```

So for the 8 possible vacuum environment configurations, the average performance score is 1.5.

3 Chp 3, Ex 3.2

a. The starting point for this puzzle is the maze's center, where the robot is pointing north. The robot may face north, south, east, or west and move a predefined distance as commanded by the operator. It will, however, come to a halt if it gets too close to a wall in order to prevent colliding with it. Because the robot can go a given distance and face north, south, east, or west, it can travel a certain distance north, south, east, or west. Given these alternatives, the robot will either stop when it reaches a wall to avoid striking it. There are five possible actions for the robot: face north, face south, face east, face west, and move. Unless it hits a wall, it can continue to travel in the direction it is facing. In such instance, it will come to a halt before colliding with the wall. So, for each action it performs, there are two possible outcomes: the robot may either advance in the direction it is facing or it cannot because it has hit a wall. There are 4n potential world states given a location of size n, because the robot can move in 4 different directions.

Initial state: Maze center, pointing north. Following the format, (In (start), face(n))

Successor function: (In (state), face(x)), where x = n, s, e, or w

State space: There are 4n potential world states since there are n different locations and 4 possible routes to go in.

Actions: face north, face south, face east, face west, move

Goal: reach any location outside of the maze

b. Because the robot only turns at corridor intersections (i), rather than anywhere in the state (n), the state space is now 4i.

In addition, because the intersections are clearly part of the state's locations, i is included in n. Since t he robot only changes direction at i, it must be traveling at every other n except i. Finally, the only two potential outcomes of the robot's movement are included (move ahead or halt when it reaches a wall), resulting in a total state space of 4i + 2(n − i).

c. The robot can now travel in any direction unaided and the only action that needs to be done is now turning, as a result, the state space can only be represented by i. Another consequence of excluding all of these operations is that we no longer need to keep track of the robot's orientation because it now executes all of its tasks without the user's input.

d.

1. The robot's sensors are non-erroneous and the robot never hits a wall.
2. If the maze has any dead ends, the robot would be stuck due to the fact that it can only turn at intersections.
3. The robot can turn, but cannot move diagonally.

4 Chp 3, Ex 3.3

a. This search problem consists of the following components:

Goal: place both people, i.e. p1 and p2 in the same city i, in the lowest possible time T.

Define dist(i, j) as the straight-line distance between two cities i and j.

Problem:

- States: the pair of cities i, j, that p1 and p2, respectively, are currently located in. i and j may represent the same city.
- Actions: moving p1 to city i' and p2 to city j', incurring an addition to T of max(dist(i, i'), dist(j, j'))

Solution:

The solution is two equal-length sequences of cities, one for p1 and one for p2, in the form $i \to j \to ... \to k \to l \to m$ and a $\to b \to ... \to c \to d \to m$.

The final city must be in each sequence, and the sequences must not share any other cities in the same index of the sequence. Additionally, the solution must minimize the following function (both of length n):

ds1 = distances traveled by p1 in sequence 1

ds2 = distances traveled by p2 in sequence 2

get the largest distance for each position

dist = map(max, zip(ds1, ds2))

This results in the two people being in the same place with the shortest overall distance traveled.

b. The heuristic D(i, j)/2 is admissible. In the ideal case, both people proceed toward each other at equal pace, resulting in twice the distance reduction compared to if one of them had simply walked to the other. Therefore, the solution becomes more optimal as it approaches the heuristic.

c. There are completely connected maps for which no solution exists. The easiest example is a two node graph. As both have to move at once, they will simply swap places and can never be in the same city.

d In specific cases all solutions require a person to visit the same node twice, e.g.

```
(A)---( )---( )---( )---( )---(C)---(B)---( )---( )
                                      \          /
                                        \      /
                                           ( )
```

A and B above can meet at C, but this requires B to go around the loop and return to B. All other solutions (with more steps) would require either backtracking or for A to move around the loop and return to its entry point of it as well.