Robin Mehta
Introduction to AI: Homework 1

1. Characterizing An Agent's Environment

Crossword Puzzle Solver

Tests at a crossword puzzle solution to determine if it is solved correctly or not.

A) PEAS Description:

Performance Measure: Whether the words' characters fit in the spaces, and whether the words correctly answer the riddles associated.

Environment: The computer program that's running the crossword solver, or the computer itself (or tablet or phone).

Actuators: Screen display of a crossword solution's success or failure.

Sensors: A keyboard to create potential solutions to test.

B) Description of the Task Environment:

Fully observable: The crossword solver can observe the entire state of the solution at any point in time. If the environment is classified as the entire computer though, it would be partially observable because the crossword solver would not have access to the rest of the computer's hard drive.

Deterministic: The next state of the crossword puzzle solver is completely determined by whether or not the agent relays a success or fail response.

Sequential: The crossword solver is a sequential mechanism where the actions carried out by the agent occur in order. The agent's performance is not dependent on any episodes it perceives.

Static: The computer program is unchanged while the agent is deliberating a solution.

Discrete: The number of percepts and actions are limited and clearly defined.

Single Agent: There is a single agent operating by itself in the computer program to update the environment with the success or failure of a solution.

Roomba

A) PEAS Description:

Performance Measure: Whether the environment is clean, whether the roomba stops cleaning when the environment is clean (instead of infinite looping), whether the roomba continues cleaning if a mess is made in the middle of its progress, and whether the roomba can change directions upon encountering obstacles.

Environment: The room in which the roomba is in.

Actuators: The vacuum mechanism of the roomba, and the wheels that move the roomba to new spots to clean

Sensors: A camera lense that takes in visuals to detect dirt on the floor. Also, a bumper pick up on nearby obstacles.

B) Description of the Task Environment:

Partially observable: The roomba cannot observe the state of the room as a whole; it can only observe the parts of the room that it is cleaning and then move on.

Stochastic: The environment has aspects beyond the control of the agent (if something interferes in the environment and it becomes dirty again). The roomba has to have utility functions to adapt to these changes in the environment.

Episodic: A roomba's agents can be divided into episodes of cleaning, and episodes of moving. It doesn't have a sequential set of actions to carry out that can trickle down if interrupted early in the chain.

Dynamic: The room in which the roomba is cleaning does change (it gets cleaner) and it can change in other ways as well, if people or pets enter the room.

Continuous: There is a large range of percepts and actions that a roomba can encounter: any change to the room can affect a roomba's performance.

Multi Agent: There is are multiple agents operating in the roomba to update the environment (1 - vacuuming, and 2 - moving to a new spot)

Netflix Automatic Recommender System

A) PEAS Description:

Performance Measure: Whether the person likes a recommended show, would share/suggest to friends

Environment: The television that hosts Netflix, or just Netflix itself.

Actuators: The screen display of movies being updated as they are recommended by the system.

Sensors: User data as inputs, and a record of shows a user is clicking on, starting to watch, and finishing watching

B) Description of the Task Environment:

Fully observable: Netflix's recommendation system is fully observable of Netflix as a program. If one considers the environment as the entire television set, then one could argue that Netflix's Recommendation System is partially observable in that scenario.

Stochastic: Consider a user watching Netflix. A user can change the environment by watching, stopping, finishing, liking, reviewing, or recommending a show. There are many variables that can change the the recommendation system's environment, each resulting in multiple possible successor states, changing with different probabilities which shows are recommended to the user.

Sequential: Netflix's recommendation system builds upon itself, as it observes more sensory input from the user. If a user watches and likes one show, that knowledge will be used in the future for the agent to recommend new shows.

Dynamic: While the recommendation system is deliberating, the environment is changing to continually update new recommendations based on new input.

Discrete: There is a limited number of discrete percepts and actions for Netflix's recommendation system to encounter: all stemming from a user's interaction with Netflix.

Single Agent: There is a single agent operating to update a view with new Netflix recommendations

Home Alarm

A) PEAS Description:

Performance Measure: Whether the alarm rings on a home break-in, and doesn't ring otherwise.

Environment: The home

Actuators: The alarm that rings in the home

Sensors: A number-pad for users to enter a code upon opening a door.

B) Description of the Task Environment:

Partially observable: A home alarm is partially observable of the home it is in. It cannot observe anything about the home other than an incorrect code entered in its number pad.

Deterministic: The state of the alarm is entirely dependent on the previous state and the actuator's actions. An actuator can either deem a code correct or incorrect, and that directly determines the next state of the environment (the alarm goes off in the house).

Episodic: Over the course of time, the actuator's actions don't affect future actions. They are instead broken into episodes, where the alarm is either deliberating a home-intruder or not. If the alarm goes off once, it doesn't have a system in place to affect the probability of it occurring again in the future (that would be sequential).

Dynamic: While the alarm system is deliberating, the environment (home) can change in a myriad of ways.

Discrete: There is a limited number of discrete percepts and actions for a home alarm's system to encounter. Percepts: A door opening, a code entered, a window opening. Actions: Ring the alarm, remain quiet, or stop the alarm.

Single Agent: There is a single agent operating to ring or silence a home alarm.

Boston Dynamics BigDog

A) PEAS Description:

Performance Measure: Whether the BigDog doesn't slip on ice, can walk straight, can carry a heavy load, can retain balance after being pushed

Environment: Any rough terrain

Actuators: The physical object that moves through the terrain (perhaps, the legs that leave footprints)

Sensors: Locomotion sensors to retain balance and move through rough terrain

B) Description of the Task Environment:

Partially observable: BigDog can only observe sensory input from its locomotion sensors of the terrain. It really does not have a full understanding of the terrain it is in.

Stochastic: The state of BigDog is not entirely dependent on the previous state and the actuator's actions, because the environment is changing so rapidly. There are many possible actions the actuators can perform, with different probabilities, based on environmental factors.

Sequential: BigDog's actions can affect future actions. Example: If it slips on ice and breaks a leg, then all future actions are affected.

Dynamic: The terrain can change at any given moment.

Continuous: There is a large range of values of percepts a BigDog can encounter and actions it can take.

Multi Agent: There are multiple agents working together to update the environment (take steps, move, etc).

Siri

A) PEAS Description:

Performance Measure: Whether Siri interprets voice correctly, and provides a valid answer.

Environment: iPhone

Actuators: Sound bytes of voice spoken back from iPhone, and text displayed on the iPhone's screen.

Sensors: A microphone to gather sound inputs.

B) Description of the Task Environment:

Partially observable: Siri as a program does not have access to private, encrypted information within the iPhone.

Stochastic: Siri's actions can vary by probability, and has utility functions to make guesses with inputs it cannot interpret accurately.

Episodic: Siri's actions do not affect future actions. Once it is finished interacting with a user, those interactions don't compound during the next interaction.

Semi: The iPhone's hardware is generally static and unchanging, but the software and memory is constantly dynamic and changing as a user uses it.

Continuous: There is a large range of values of percepts Siri's microphone can encounter and actions it can take based on that.

Multi Agent: There are multiple agents working together to update the environment (speaker, text on a screen).

Paro Therapeutic Robot

A) PEAS Description:

Performance Measure: Whether a patient is comforted by Paro

Environment: a hospital or extended care facility

Actuators: the physical movements of a Paro, and the sounds it makes

Sensors: A microphone, tactile sensors, camera lense for visual inputs

B) Description of the Task Environment:

Partially observable: Paro cannot observe everything happening in the hospital or extended care facility.

Stochastic: Paro's actions can vary by probability, and has utility functions to make guesses with inputs it cannot interpret accurately.

Sequential: Paro's actions do affect future actions. As it interacts with a user, the knowledge it learns compounds over time to improve upon future interactions.

Dynamic: A hospital is a constantly changing environment.

Continuous: There is a large range of values of percepts Paro's sensors can encounter and actions it can take based on that.

Multi Agent: There are multiple agents working together to update the environment (speaker, physical movements).

Problem 2, Part 5

1) FA-bot PEAS Description:

Performance Measure: Whether the bot can successfully weed and water each square without running out of water or power.

Environment: The farm

Actuators: the methods that carry out weeding, watering, moving, etc. to update the farm

Sensors: The percept methods that sense each square's boolean values that indicate the need for watering and weeding.

2) Rational behavior:

The definition of rational behavior is behavior that works towards getting all squares in the farm watered and weeded with the resources given.

3) Is the simple reflex agent perfectly rational?

The simple reflex agent is not perfectly rational for this environment, because it is not the optimal solution to achieve the rational agent's goal. The state reflex agent from Part 2, however, is perfectly rational because it is the most optimal of the four agents, and succeeds for each environment.

4) Outcomes of each agent:

Simple Reflex Agent

Env. #	Succeeded?	# Moves	Water	Power	# Left to water	# Left to weed
1	yes	27	1	73	0	0
2	no	25	5	0	1	1

3	yes	30	3	70	0	0
4	yes	42	5	58	0	0
5	no	100	7	0	3	1

State Reflex Agent

Env. #	Succeeded?	# Moves	Water	Power	# Left to water	# Left to weed
1	yes	23	1	77	0	0
2	yes	7	4	18	0	0
3	yes	27	3	73	0	0
4	yes	37	5	63	0	0
5	yes	47	4	53	0	0

Random Reflex Agent (averages)

Env. #	Succeeded?	# Moves	Water	Power	# Left to water	# Left to weed
1	yes	65.2	1	34.8	0	0
2	yes	17	3.2	8	0.2	0.2
3	yes	64.4	3.2	35.6	0.2	0.2
4	no	89	6	11	0.8	0.2
5	no	100	5.8	0	1.8	1.2

Better Reflex Agent

Env. #	Succeeded?	# Moves	Water	Power	# Left to water	# Left to weed
1	yes	25	1	75	0	0
2	no	25	5	0	1	1
3	yes	30	3	70	0	0
4	yes	40	5	60	0	0
5	no	100	7	0	3	1

5. Which agent performed best?

The state reflex agent outperformed every other agent by succeeded for each environment. This is because it was made to optimize for its rational agent: watering and weeding the farm while using the least amount of resources. My randomized simple agent outperforms the simple reflex agent in environment 2, because of the grid arrangement that is worse with a clock-wise movement cycle than a random movement system.

6. In which environments did agent from Part 4 outperform agent from Part 1?

My better reflex agent outperformed my simple reflex agent in environments 1 and 4, and no worse in all other environments. I structured my better reflex agent to keep track of the number of barriers hit, the number of successful moves, and the direction relative to the starting direction (simply, the number of barriers % 4). I noticed that in these environments, if the number of successful moves % 4 = 0, and direction = 3, then there were a few squares that were repeatedly being visited, and didn't need to be checked for water and weeds again. My better reflex agent outperformed the simple reflex agent by succeeding in less number of moves and more resources left over for environments 1 and 4.

7. Weeding Strategy: 1 unit to sense, 5 units to weed

It makes most sense to sense for weed first for each square, and weed only if necessary.

8. Weeding Strategy: 5 units to sense, 1 unit to weed

It makes most sense to weed every square and never sense if anything needs to be weeded.

9. Indeterminism

The state-based agent from Part 2 would no longer be a perfect rational agent if it could only weed 75% of the time. It would likely still be optimal over the simple reflex agent, but not perfectly successful for every environment.

10. Modifying the indeterministic agent

I would modify the indeterministic, state-based agent to add a “weeding_unsuccessful” percept, which can then return a “weed” action. That way, on the 25% chance that weeding was unsuccessful, it would retry indefinitely until it weeds.

Agents.py Code Snippets:

robin_mehta_agents.pdf

robin_mehta_agents.pdf ×

```python
# Robin Mehta
# EECS 492 Homework 1

import world
import numpy
from numpy.random import randint

class simple_reflex_agent:
    def __init__(self):
        # Agent has no state, so nothing to do here
        return

    # Given a valid percept (a string), return a valid action (also as a string)
    def choose_action(self, percept):
        ##### Implement this function #####
        if percept == "start":
            return "sense_water"
        if percept == "needs_watering":
            return "water"
        if percept == "watering_succeeded":
            return "sense_weed"
        if percept == "does_not_need_watering":
            return "sense_weed"
        if percept == "needs_weeding":
            return "weed"
        if percept == "does_not_need_weeding":
            return "move"
        if percept == "weeding_succeeded":
            return "move"
        if percept == "move_succeeded":
            return "sense_water"
        if percept == "hit_barrier":
            return "move"

        # Returning an arbitrary action so don't get import errors if function not written
        # You may remove this when you've finished implementing
        return 'water'
        ###################################


class state_reflex_agent:
    def __init__(self, worldstate):
        # This agent has a state, so wants to do BFS on the world
        # self.plan is a list of moves for the agent to take through the world
        # self.position is the position in the list
        self.plan, plan_exists = world.BFS(worldstate)
        if not plan_exists:
            raise RuntimeError('Plan does not exist or BFS iteration limit exceeded')
        self.position = 0
```

```
50
51      # Given a valid percept (a string), return a valid action (also as a string)
52      def choose_action(self, percept):
53          if (percept == "start") or (percept == "move_succeeded"):
54              return "sense_water"
55
56          if percept == "needs_watering":
57              return "water"
58
59          if (percept == "watering_succeeded") or (percept == "does_not_need_watering"):
60              return "sense_weed"
61
62          if percept == "needs_weeding":
63              return "weed"
64
65          if (percept == "does_not_need_weeding") or (percept == "weeding_succeeded"):
66
67              if self.plan[self.position] == "move_north":
68                  self.position = self.position + 1
69                  return "move_north"
70
71              elif self.plan[self.position] == "move_east":
72                  self.position = self.position + 1
73                  return "move_east"
74
75              elif self.plan[self.position] == "move_south":
76                  self.position = self.position + 1
77                  return "move_south"
78
79              else:
80                  self.position = self.position + 1
81
82                  return "move_west"
83
84          if percept == "hit_barrier":
85              return ""
86
87          ################################
88
89  class random_reflex_agent:
90      def __init__(self):
91          # Agent has no state, so nothing to do here
92          return
93
94      # Given a valid percept (a string), return a valid action (also as a string)
95      def choose_action(self, percept):
96          #### Implement this function ####
97          if (percept == "start") or (percept == "move_succeeded"):
98              return "sense_water"
```

```
 99
100         if percept == "needs_watering":
101             return "water"
102
103         if (percept == "watering_succeeded") or (percept == "does_not_need_watering"):
104             return "sense_weed"
105
106         if percept == "needs_weeding":
107             return "weed"
108
109         if (percept == "does_not_need_weeding") or (percept == "weeding_succeeded") or (percept == "
110
111             randomInt = numpy.random.randint(0, 4)
112
113             if randomInt == 0:
114                 return "move_north"
115
116             elif randomInt == 1:
117                 return "move_east"
118
119             elif randomInt == 2:
120                 return "move_south"
121
122             else:
123                 return "move_west"
124
125         # Returning an arbitrary action so don't get import errors if function not written
126         # You may remove this when you've finished implementing
127         return 'water'
128         ################################
129
130 class better_reflex_agent:
131     def __init__(self):
132         # Agent cannot see the world in advance
133         # However, you may initialize any number of your own state variables here
134
135         self.num_barriers = 0
136         self.direction = 0
137         self.num_successful_moves = 0
138
139         return
140
141     # Given a valid percept (a string), return a valid action (also as a string)
142     def choose_action(self, percept):
143         if percept == "start":
144             return "sense_water"
145         if percept == "needs_watering":
146             return "water"
```

```
        if percept == "watering_succeeded":
            return "sense_weed"
        if percept == "does_not_need_watering":
            return "sense_weed"
        if percept == "needs_weeding":
            return "weed"
        if percept == "does_not_need_weeding":
            return "move"
        if percept == "weeding_succeeded":
            return "move"
        if percept == "move_succeeded":
            self.num_successful_moves = self.num_successful_moves + 1
            if ((self.num_successful_moves % 4) == 0) and ((self.num_barriers % 4) == 3):
                return "move"
            else:
                return "sense_water"
        if percept == "hit_barrier":
            self.num_barriers = self.num_barriers + 1
            self.direction = self.direction + 1
            self.direction = self.direction % 4
            return "move"

        # Returning an arbitrary action so don't get import errors if function not written
        # You may remove this when you've finished implementing
        return 'water'
        ################################


```