Artificial Intelligence Assignment #1

20175183 Yura Choi

Problem 1 table lookup, simple reflex, goal-based, utility-based

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, suggestions, corrections	Keyboard entry

Figure 2.5 Examples of agent types and their PEAS descriptions.

- **Medical diagnosis system**:

  Table lookup agent will be appropriate as it should diagnosis based on some pre-defined medical information according to patient's symptoms.

- **Satellite image analysis system**:

  Simple reflex agent is more possible. Satellite image analysis system does not have to consider the previous sequence of percepts or the internal state of the agent.

- **Part-picking robot**

  Simple reflex agent.

- **Refinery controller**

  Utility based.

- **Interactive English tutor**

  Goal- based agent to maximize the student's score on the test

Problem 2

Agent is a combination of architecture and program that take an action based on perceived environment. An actuator is the acting part of the agent, and sensors perceive the environment.

Agent function is a map from any percept sequence to the action of the agent. It is usually represented by the table.

To be implemented, agent must be programmed with respect to agent function, and it is **agent program**.

A rational agent selects action that optimizes an expected output environment. **Rationality**, in the same context, is the property to maximize the 'expected' performance (perfection is different from this, it is hard) of the agent by using the agent's prior knowledge on the environment and percept sequence up to date.

Autonomy is the ability of an agent acting independent to the pre-defined knowledge. It means that the agent can make its own decision, learning from experiences of environment.

Reflex agent is an agent which selects actions from condition-action rule on current environment, without considering percept history.

Model-based agent uses the model of the world is called as a model-based agent. The knowledge about "how the world works" is called a model of the world.

Goal-based agents are model-based agents which sorts goal information that describes situations

Utility-based agent is an agent that uses an explicit utility function that maximizes the expected utility.

Learning agent is an agent that improves its behavior based on its experiences and learning.

Problem 3

```
# Goal-based agent

function GOAL-BASED-AGENT(percept) returns action:
    persistent:
    state, what the current agent sees as the world state
    model, a description detailing how the next state is a result of the curre
nt state and StopAsyncIteration
    state, what the current agent sees as the world state
    goals, a set of goals the agent needs to accomplish (similar to a reflex a
gent's rules)
    action, the action that most recently occurred and is initially null
    state UPDATE-STATE(state, action, percept, model)
    action BEST-ACTION(goals, state)
    return action

function BEST-ACTION:
    determines the action that most furthers the agent towards fulfilling its
gloals


# Utility-based agent

function UTILITY-BASED-AGENT (percept) returns action:
    persistent:
    state, what the current agent sees as the world state
    goals, a set of goals the agent needs to accomplish (similar to a reflex a
gent's rules)
    utility, internal performance measurement
    action, the action that most recently occurred and is initially null
    state UPDATE-STATE(state, action, percept, model)
    utility UTILITY-FUNCTION(goals, state)
    action BEST-ACTION(goals, state, utility)
    return action

UTILITY-
function calculates the utility of the possible actions given the state and go
als.
BEST-
ACTION is updated from above to take the utility of each action into account w
hen choosing the best action.
```

Problem 4

a) A ping pong game should have two players. This is an expression for the probability of winning a game in a tennis match, which is derived from the assumption that the outcome of each point is identically and independently distributed. Artificial intelligence can be used to discover the fast-paced game of ping pong.

b) No, it is not possible to drive in the center of using artificial intelligence literature. The traffic intensity changes dynamically and hence it has to think and act accordingly. So it is not possible. It is possible by artificial intelligence to drive along a curving mountain road because the road map will be clear and so the instructions can be programmed clearly.

c) No. No robot can move and find supposed objects efficiently. It efficiency in identifying wide variety of objects using vision is not up to the mark and grasping the objects which are squishable is to be effectively managed.

d) Yes, AI literature is used to buy a week's worth of groceries on the web. Groceries ordered in five minutes on the web and delivered to your door are worth more than groceries on a supermarket shelf that you have to fetch yourself. Online shopping helps us to select items and bring the groceries to our front steps.

e) Yes. AI can learn to play games, by various ways e.g.) reinforcement learning. Game provides an excellent test bed for investigating potentials.

f) Yes. some mathematical theorems currently solved by computers. Theorem proving is one of research fields in AI, experimental mathematics. Google AI system proves over 1200 mathematical theories. Some exceptions are included in several theorems like logical, geometry, and algebra.

g) No. Google AI introduced poem generator. Intelligent agents might be trained by set of fun proses and learn to generate similar ones. However, it may be hard for AI to understand what point makes its proses "funny".

h) Yes. Intelligent agent may perform better at combining vast amount of legal information. Indeed, there was a competition between 20 human lawyers and artificial intelligence lawyer at 2018 and intelligent agent defeated human lawyers.

i) Yes, it is possible to translate spoken English into spoken Swedish in real time by deciphering chunks of language rather than breaking down the structure.

j) Yes, because doctors perform delicate brain surgery operations in which robots assist them. AI can be programmed to take input of problem situation, such as structure and location of the tumor in the brain and manipulate the robot's action delicately.

Problem 5

```
import numpy as np
import sys

ACTIONS = ((0, "Go Forward"),
           (1, "Turn Right"),
           (2, "Turn Left"),
           (3, "Suck Dirt"),
           (4, "Turn Off"),
           (-1, "Break"),)

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


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 circl
ing edge)
        return ACTIONS[np.random.randint(3)]


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_wall()

        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_wall()

    def is_wall(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 po
sition
    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()


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









#touch sensor = 1 if bump 0 otherwise


ENV_SIZE = (12, 12)
DIRT_CHANCE = 0.05
```

```
def main():
    env = VacuumEnvironment(ENV_SIZE, DIRT_CHANCE)
    agent = InternalAgent()

    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)
        turn += 1

    print (env.room[0])

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