COMP -237\_(002)

Introduction To AI

Assignment Agents

Gitansh Mittal

3031200517

Link for My Video Explanation is

<https://screencast-o-matic.com/watch/c3QZcnVOdOw>

Class Diagram

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

# -\*- coding: utf-8 -\*-

"""

Created on Sun Jun 7 11:38:46 2020

@author: mhabayeb

"""

import collections

from collections.abc import Callable

import numbers

class Thing:

"""This represents any physical object that can appear in an Environment.

You subclass Thing to get the things you want. Each thing can have a

.\_\_name\_\_ slot (used for output only)."""

def \_\_repr\_\_(self):

return '<{}>'.format(getattr(self, '\_\_name\_\_', self.\_\_class\_\_.\_\_name\_\_))

def is\_alive(self):

"""Things that are 'alive' should return true."""

return hasattr(self, 'alive') and self.alive

def show\_state(self):

"""Display the agent's internal state. Subclasses should override."""

print("I don't know how to show\_state.")

class Agent(Thing):

"""An Agent is a subclass of Thing with one required slot,

.program, which should hold a function that takes one argument, the

percept, and returns an action. (What counts as a percept or action

will depend on the specific environment in which the agent exists.)

Note that 'program' is a slot, not a method. If it were a method,

then the program could 'cheat' and look at aspects of the agent.

It's not supposed to do that: the program can only look at the

percepts. An agent program that needs a model of the world (and of

the agent itself) will have to build and maintain its own model.

There is an optional slot, .performance, which is a number giving

the performance measure of the agent in its environment."""

def \_\_init\_\_(self, program=None):

self.alive = True

self.bump = False

self.holding = []

self.performance = 0

if program is None or not isinstance(program, collections.abc.Callable):

print("Can't find a valid program for {}, falling back to default.".format(self.\_\_class\_\_.\_\_name\_\_))

def program(percept):

return eval(input('Percept={}; action? '.format(percept)))

self.program = program

def can\_grab(self, thing):

"""Return True if this agent can grab this thing.

Override for appropriate subclasses of Agent and Thing."""

return False

class Environment:

"""Abstract class representing an Environment. 'Real' Environment classes

inherit from this. Your Environment will typically need to implement:

percept: Define the percept that an agent sees.

execute\_action: Define the effects of executing an action.

Also update the agent.performance slot.

The environment keeps a list of .things and .agents (which is a subset

of .things). Each agent has a .performance slot, initialized to 0.

Each thing has a .location slot, even though some environments may not

need this."""

def \_\_init\_\_(self):

self.things = []

self.agents = []

def thing\_classes(self):

return [] # List of classes that can go into environment

def percept(self, agent):

"""Return the percept that the agent sees at this point. (Implement this.)"""

raise NotImplementedError

def execute\_action(self, agent, action):

"""Change the world to reflect this action. (Implement this.)"""

raise NotImplementedError

def default\_location(self, thing):

"""Default location to place a new thing with unspecified location."""

return None

def exogenous\_change(self):

"""If there is spontaneous change in the world, override this."""

pass

def is\_done(self):

"""By default, we're done when we can't find a live agent."""

return not any(agent.is\_alive() for agent in self.agents)

def step(self):

"""Run the environment for one time step. If the

actions and exogenous changes are independent, this method will

do. If there are interactions between them, you'll need to

override this method."""

if not self.is\_done():

actions = []

for agent in self.agents:

if agent.alive:

actions.append(agent.program(self.percept(agent)))

else:

actions.append("")

for (agent, action) in zip(self.agents, actions):

self.execute\_action(agent, action)

self.exogenous\_change()

def run(self, steps=1000):

"""Run the Environment for given number of time steps."""

for step in range(steps):

if self.is\_done():

return

self.step()

def list\_things\_at(self, location, tclass=Thing):

"""Return all things exactly at a given location."""

if isinstance(location, numbers.Number):

return [thing for thing in self.things

if thing.location == location and isinstance(thing, tclass)]

return [thing for thing in self.things

if all(x == y for x, y in zip(thing.location, location)) and isinstance(thing, tclass)]

def some\_things\_at(self, location, tclass=Thing):

"""Return true if at least one of the things at location

is an instance of class tclass (or a subclass)."""

return self.list\_things\_at(location, tclass) != []

def add\_thing(self, thing, location=None):

"""Add a thing to the environment, setting its location. For

convenience, if thing is an agent program we make a new agent

for it. (Shouldn't need to override this.)"""

if not isinstance(thing, Thing):

thing = Agent(thing)

if thing in self.things:

print("Can't add the same thing twice")

else:

thing.location = location if location is not None else self.default\_location(thing)

self.things.append(thing)

if isinstance(thing, Agent):

thing.performance = 0

self.agents.append(thing)

def delete\_thing(self, thing):

"""Remove a thing from the environment."""

try:

self.things.remove(thing)

except ValueError as e:

print(e)

print(" in Environment delete\_thing")

print(" Thing to be removed: {} at {}".format(thing, thing.location))

print(" from list: {}".format([(thing, thing.location) for thing in self.things]))

if thing in self.agents:

self.agents.remove(thing)

############

class BlindDog(Agent):

def eat(self, thing):

print("Dog: Ate food at {}.".format(self.location))

def drink(self, thing):

print("Dog: Drank water at {}.".format( self.location))

def bark(self, thing):

print("Dog: Bark at person {}.".format( self.location))

dog = BlindDog()

#############

print(dog.alive)

############

class Food(Thing):

pass

class Water(Thing):

pass

class Tree(Thing):

pass

#2

class Person(Thing):

pass

class Park(Environment):

def percept(self, agent):

'''return a list of things that are in our agent's location'''

things = self.list\_things\_at(agent.location)

return things

def execute\_action(self, agent, action):

'''changes the state of the environment based on what the agent does.'''

if action == "move down":

print('{} decided to {} at location: {}'.format(str(agent)[1:-1], action, agent.location))

agent.movedown()

elif action == "eat":

items = self.list\_things\_at(agent.location, tclass=Food)

if len(items) != 0:

if agent.eat(items[0]): #Have the dog eat the first item

print('{} ate {} at location: {}'

.format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))

self.delete\_thing(items[0]) #Delete it from the Park after.

elif action == "drink":

items = self.list\_things\_at(agent.location, tclass=Water)

if len(items) != 0:

if agent.drink(items[0]): #Have the dog drink the first item

print('{} drank {} at location: {}'

.format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))

self.delete\_thing(items[0]) #Delete it from the Park after.

elif action == "bark":

items = self.list\_things\_at(agent.location, tclass=Person)

if len(items) != 0:

if agent.bark(items[0]): #Have the dog drink the first item

print('{} bark {} at location: {}'

.format(str(agent)[1:-1], str(items[0])[1:-1], agent.location))

self.delete\_thing(items[0])#Delete it from the Park after.

def is\_done(self):

'''By default, we're done when we can't find a live agent,

but to prevent killing our cute dog, we will stop before itself - when there is no more food or water'''

no\_edibles = not any(isinstance(thing, Food) or isinstance(thing, Water) or isinstance(thing, Person) for thing in self.things)

dead\_agents = not any(agent.is\_alive() for agent in self.agents)

return dead\_agents or no\_edibles

#########

# So we defined everything now we can run a program#

class BlindDog(Agent):

location = 1

def movedown(self):

self.location += 1

def eat(self, thing):

'''returns True upon success or False otherwise'''

if isinstance(thing, Food):

return True

return False

def drink(self, thing):

''' returns True upon success or False otherwise'''

if isinstance(thing, Water):

return True

return False

def bark(self, thing):

''' returns True upon success or False otherwise'''

if isinstance(thing, Person):

return True

return False

##########

def program(percepts):

'''Returns an action based on the dog's percepts'''

for p in percepts:

if isinstance(p, Food):

return 'eat'

elif isinstance(p, Water):

return 'drink'

elif isinstance(p, Person):

return 'bark'

return 'move down'

#Now its time to implement a program module for our dog. A program controls how the dog acts upon its environment. Our program will be very simple, and is shown in the table below.

# Percept: Feel Food Feel Water Feel Nothing

#Action: eat drink move down

#####

#Assignment

park = Park()

dog = BlindDog(program)

dogfood = Food()

water = Water()

gitansh = Person()

mittal = Person()

park.add\_thing(dog, 1)

park.add\_thing(dogfood, 9)

park.add\_thing(gitansh, 3)

park.add\_thing(mittal, 12)

park.run(18)