ExpNo 1 :Developing AI Agent with PEAS Description

Name: Saravanan N

Register Number/Staff Id: TSML006

AIM:

To find the PEAS description for the given AI problem and develop an AI agent.

Theory

Medicine prescribing agent:

Such this agent prescribes medicine for fever (greater than 98.5 degrees) which we consider here as unhealthy, by the user temperature input, and another environment is rooms in the hospital (two rooms). This agent has to consider two factors one is room location and an unhealthy patient in a random room, the agent has to move from one room to another to check and treat the unhealthy person. The performance of the agent is calculated by incrementing performance and each time after treating in one room again it has to check another room so that the movement causes the agent to reduce its performance. Hence, agents prescribe medicine to unhealthy.

PEAS DESCRIPTION:

Agent Type	Performance	Environment	Actuators	Sensors
Medicine prescribing agent	**Treating unhealthy, agent movement**	**Rooms, Patient**	**Medicine, Treatment**	**Location, Temperature of patient**

DESIGN STEPS

STEP 1:Identifying the input:

Temperature from patients, Location.

STEP 2:Identifying the output:

Prescribe medicine if the patient in a random has a fever.

STEP 3:Developing the PEAS description:

PEAS description is developed by the performance, environment, actuators, and sensors in an agent.

STEP 4:Implementing the AI agent:

Treat unhealthy patients in each room. And check for the unhealthy patients in random room

STEP 5:

Name: praveen . c k

Register Number: 212222243003

```
import random
import time
class Thing:
"""
This represents any physical object that can appear in an Environment.
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 """
def init(self, program=None):
self.alive = True
self.performance = 0
self.program = program
def can_grab(self, thing):
"""Return True if this agent can grab this thing. Override for appr
return False
In [8]:
def TableDrivenAgentProgram(table):
"""



This agent selects an action based on the percept sequence. It is pract
To customize it, provide as table a dictionary of all
{percept_sequence:action} pairs. """
percepts = []


def program(percept):
action = None
percepts.append(percept)
action = table.get(tuple(percepts))
return action
return program


loc_A, loc_B = (0,0), (1,0) # The two locations for the Vaccum cleaning
```

```
def TableDrivenVaccumAgent():
"""
Tabular approach towards Vaccum cleaning
"""


table = {
((loc_A, "Clean"),): "Right",
((loc_A, "Dirty"),): "Suck",
((loc_B, "Clean"),): "Left",
((loc_B, "Dirty"),): "Suck",
((loc_A, "Dirty"), (loc_A, "Clean")): "Right",
((loc_A, "Clean"), (loc_B, "Dirty")): "Suck",
((loc_B, "Clean"), (loc_A, "Dirty")): "Suck",
20/02/2024, 22:25 VACCUM-Copy1
localhost:8888/nbconvert/html/Downloads/VACCUM-Copy1.ipynb?download=false 2/4
((loc_B, "Dirty"), (loc_B, "Clean")): "Left",
((loc_A, "Dirty"), (loc_A, "Clean"), (loc_B, "Dirty")): "Suck",
((loc_B, "Dirty"), (loc_B, "Clean"), (loc_A, "Dirty")): "Suck",
}
return Agent(TableDrivenAgentProgram(table))


class Environment:
"""Abstract class representing an Environment. 'Real' Environment classe
percept: Define the percept that an agent sees. execute_action: Def
Also update the agent.performance slot.
The environment keeps a list of .things and .agents (which is a subset
Each thing has a .location slot, even though some environments may not
def init(self):
self.things = []
self.agents = []


def percept(self, agent):
"""Return the percept that the agent sees at this point. (Implement
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 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 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)
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 add_thing(self, thing, location=None):
"""Add a thing to the environment, setting its location. For conven
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.de
self.things.append(thing)
if isinstance(thing, Agent):
20/02/2024, 22:25 VACCUM-Copy1
localhost:8888/nbconvert/html/Downloads/VACCUM-Copy1.ipynb?download=false 3/4
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.locat
print(" from list: {}".format([(thing, thing.location) for thing
if thing in self.agents:
self.agents.remove(thing)


class TrivialVaccumEnvironment(Environment):
"""This environment has two locations, A and B. Each can be clean or di
def init(self):
super().init()
#loc_A, loc_B = (0,0), (1,0) # The two locations for the Vaccum clea
self.status = {loc_A: random.choice(["Clean", "Dirty"]), loc_B: rand
def thing_classes(self):
return [TableDrivenVaccumAgent]
def percept(self, agent):
"""Returns the agent's location, and the location status (Dirty/Cle
return agent.location, self.status[agent.location]
def execute_action(self, agent, action):
"""Change agent's location and/or location's status; track performa
if action == "Right":
agent.location = loc_B
agent.performance -= 1
elif action == "Left":
agent.location = loc_A
agent.performance -= 1
elif action == "Suck":


if self.status[agent.location] == "Dirty":
agent.performance += 10
self.status[agent.location] = "Clean"
def default_location(self, thing):


return random.choice([loc_A, loc_B])


if name == "main":


agent = TableDrivenVaccumAgent()
environment = TrivialVaccumEnvironment()
#print(environment)
```

```
environment.add_thing(agent)
print(environment.status)
environment.run(steps=10)
print(environment.status)
print(agent.performance)
```

OUTPUT:

Measure the performance parameters: For each treatment performance incremented, for each movement performance decremented