Lab 01 - Intelligent Agents

CO 541 Artificial Intelligence

Department of Computer Engineering, University of Peradeniya

December 2, 2022

1 Objective

The ”agent” paradigm has rapidly become one of the major conceptual frameworks for understanding and structuring research activities within Artificial Intelligence (AI). In addition, many software frameworks are being introduced and used both in research and industry to control the complexity of sophisticated distributed software systems where the nodes exhibit behaviors often associated with intelligent behavior.

The purpose with this lab is to introduce the agent concept by allowing you to familiarize yourself with a software agent environment and world simulator. You will program an agent and implement its behaviour to provide cleaning capabilities in the vacuum cleaning world. The behaviour will then be compared to some simple reference agents.

2 Introduction

2.1 What is Artificial Intelligence?

AI is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable[1].

> **A slogan for AI:** Whatever a person can do, he should be able to make a computer do for him.
>
> ---
>
> *John McCarthy (A founding father of AI)*

If you are interested in exploring more about the AI of Philosophy, you can visit and read the following page for more information like questions and answers related to AI: http://jmc.stanford.edu/artificial-intelligence/index.html

2.2 What is an Intelligent Agent?

An idea has no value until it is put into action and, in the world of AI, an intelligent agent (IA) is the key to putting AI into action. It is an entity that can make decisions based on its environment, user input and experiences, be it by autonomously gathering information on a regular, programmed schedule or when prompted by a user in real-time. IAs are also called ‘intelligent’ because of their ability to learn during the process of performing tasks[2].

A simple IA features two main functionalities – the ability to perceive an environment using sensors and, once its artificial intelligence decides what needs to be done, to perform actions through actuators.

[1]McCarthy, J. (1970) Artificial Intelligence, Professor John McCarthy - Artificial Intelligence. Available at: http://jmc.stanford.edu/artificial-intelligence/index.html (Accessed: December 1, 2022).

[2]Choudhuri, A. (2022) What is intelligent agent in ai?, PROBEcx. MicroSourcing International Ltd. Available at: https://www.probegroup.com.au/blog/what-is-intelligent-agent-in-ai (Accessed: December 1, 2022).

In the same way that humans have sensors such as eyes, ears, touch, and taste, IAs use mediums like cameras, microphones and, natural language processing (NLP) to provide input. As for actuators, humans have hands, legs, mouths, and facial expressions to perform actions while agents use the likes of speakers, robotic arms, and screens to perform the same.

3 Your Task - Modeling a Vacuum Agent

In this lab, your task is to model different structures of agents through the **vacuum agent** example shown in Figure 1. The agent should be able to remove all the dirt from the given three locations A, B, and C with random distribution of dirt.

The role of AI is to model an **agent program** that implements the agent function (i.e., the mapping from percepts to actions). Throughout this lab, we assume that this program runs on some sort of computing device equipped with physical sensors and actuators: a.k.a. the **architecture**.

Agent = Architecture + Program

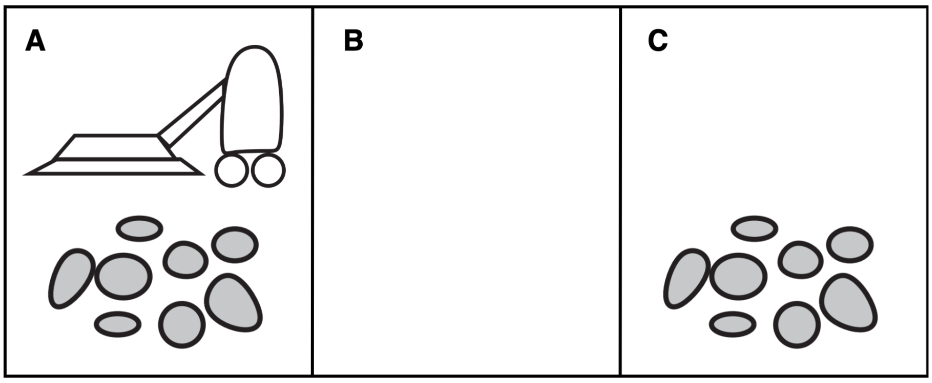

Figure 1: A vacuum-cleaner world with three locations: A, B, and C (dirt can be found anywhere)

3.1 Agent Programs

An agent program takes the current perception from the sensors as input and assigns an action to the actuators. There is a difference between an agent program and an agent function: an agent program takes the current perception as input, while an agent function takes the entire perception history.

The agent program takes only the current perception as input because nothing else can be obtained from the environment; If the agent's actions depend on the entire perceptual sequence, then the agent must remember the percept. In this lab, the following agents are discussed and the first four are modeled with the help of the given vacuum world example:

- Random Agent Program
- Table-driven Agent Program
- Simple Reflex Agent Program
- Model-based Reflex Agent Program
- Goal-based Agent Program
- Utility-based Agent Program

Before implementing any agent program, let's implement the `Thing` class to represent any physical entity (a.k.a. the blueprint for any object) that can appear in the Environment. All entities in the environment, in this case the `Agent`, must be implemented as a subclass of `Thing` to hold, a function that takes one argument, the percept and returns the corresponding action to the actuators. What counts as a perception or an action depends on the specific environment in which the agent exists. In this lab, the complete codebase for agent modeling is provided in Section 8 for your convenience.

3.1.1 Random Agent Program

As the name suggests, an action is chosen randomly without taking into account the percepts. Write an agent program to randomly choose any of the actions among **Right**, **Left**, **Suck**, and **NoOp** from a trivial vacuum environment, a.k.a. the two-state environment.

```
def RandomAgentProgram(actions):
    return """The agent must return an action at random, ignoring all percepts."""

def RandomVacuumAgent():
    return Agent("""Choose one of the actions from the vacuum
    environment randomly.""")
```

3.1.2 Table-driven Agent Program

A table-driven agent program observes the percept sequence and uses it to index into a table of actions to determine what to do. The table represents explicitly the agent function that embodies the agent program.

In the two-state vacuum world with two locations, the table consists of all the following possible states of the agent. Extend the following table to simulate the two-state vacuum world given in Figure 1. Write all the possible states as you can.

```
# Define three locations in the Vacuum World
loc_A, loc_B, loc_C = (0, 0), (1, 0), (2, 0)

# ...
table = {((loc_A, "Clean"),): "Right",
         ((loc_A, "Dirty"),): "Suck",
         ((loc_B, "Clean"),): "Right",
         ((loc_B, "Clean"),): "Left",
         ((loc_B, "Dirty"),): "Suck",
         # ...
         ((loc_A, "Clean"), (loc_A, "Clean")): "Right",
         ((loc_A, "Clean"), (loc_A, "Dirty")): "Suck",
         # ...
         ((loc_A, "Clean"), (loc_A, "Clean"), (loc_A, "Clean")): "Right",
         ((loc_A, "Clean"), (loc_A, "Clean"), (loc_A, "Dirty")): "Suck",
         # ...
         }
```

3.1.3 Simple Reflex Agent Program

A simple recursive agent program chooses actions on the basis of current perception, ignoring the rest of the perception history. These agents work on a condition-action rule (a.k.a. the situation-action rule, production or if-then rule), which tells the agent what action to trigger when a certiain condition occurs. The schematic diagram shown in Figure 2 makes this more clear.

Augment the following codebase to simulate the vacuume cleaner in the three-location problem environment:

```
def SimpleReflexAgentProgram(rules, interpret_input):
    """This agent takes action based solely on the percept of Figure 2"""
```

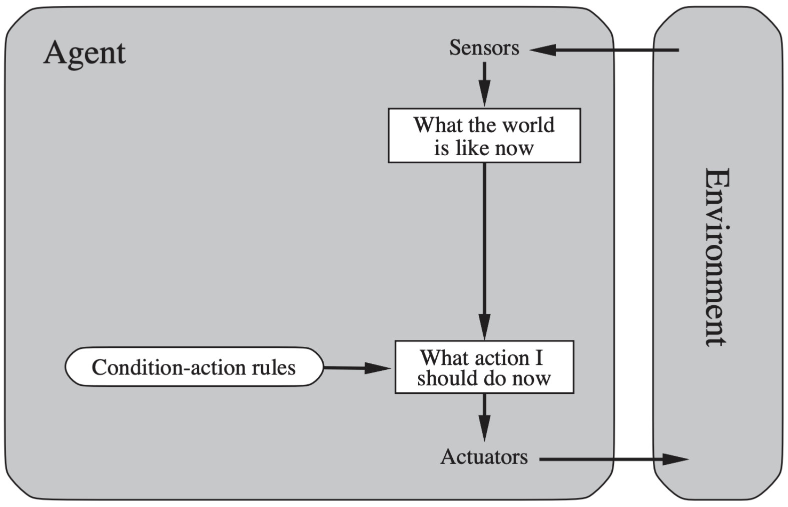

Figure 2: Schematic diagram of a simple reflex agent.

```
        def program(percept):
            state = interpret_input(percept)
            rule = rule_match(state, rules)
            action = rule.action
            return action
        return program

    def ReflexVacuumAgent():
        """A simple reflex agent for the three-state vacuum environment"""
        def program(percept):
            location, status = percept
            if status == "Dirty":
                print(percept, "Suck")
                return "Suck"
            elif location == loc_A:
                print(percept, "Right")
                return "Right"
            ...
            """Augment to simulate the three location problem environment (Figure 1)"
        return Agent(program)
```

3.1.4 Model-based Reflex Agent Program

A model-based reflex agent maintains some internal state that depends on the percept history and thereby reflects at least some unobserved aspects of the current state. In addition, it requires a model of the world, i.e., knowledge about ”how the world works”. We recommend you to study the schematic diagram shown in Figure 3 to make it more clear before the implementation. Here we need another function `update_state`, which is responsible for creating a new state description. Test the results for a model-based reflex agent program using the following codebase:

```
    def ModelBasedReflexAgentProgram(rules, update_state):
    "This agent takes action based on the percept and state. [Figure 2.12]"
    def program(percept):
        program.state = update_state(program.state, program.action, percept)
        rule = rule_match(program.state, rules)
        action = rule.action
```

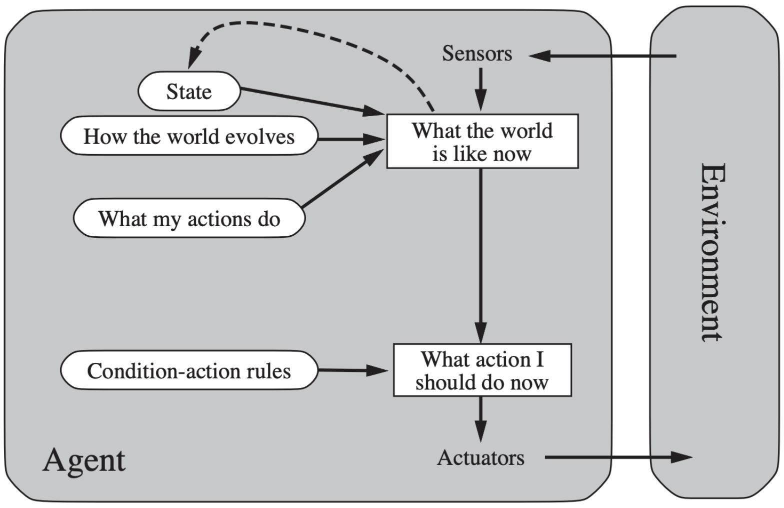

Figure 3: A model-based reflex agent.

```
        return action
    program.state = program.action = None
    return program
```

3.1.5 Goal-based Agent Program

These agents have higher capabilities than model-based reflex agents. Goal-based agents use goal information to describe desirable capabilities. This allows them to choose among various possibilities. These agents choose the best action that will enhance goal attainment.

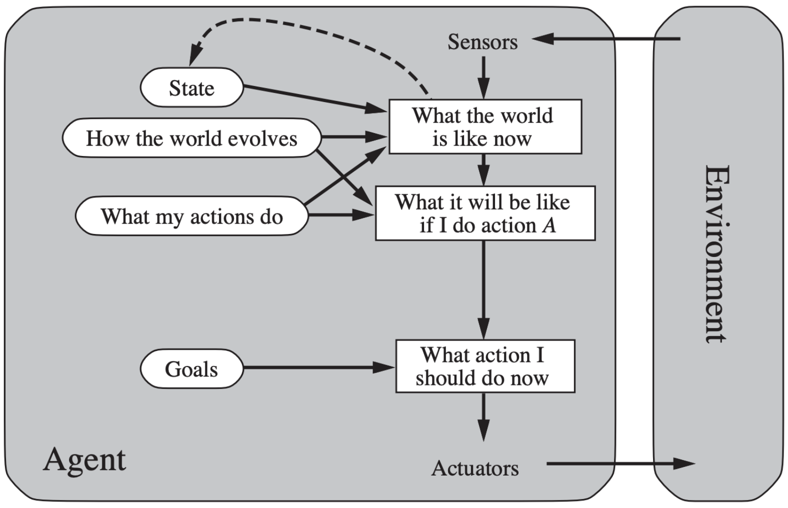
Sensors
State
How the world evolves
What the world is like now
What my actions do
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

Figure 4: A goal-based agent.

3.1.6 Utility-based Agent Program

These agents make choices based on utility. They are superior to goal-based agents because of the additional feature of utility measurement. Using a utility function, a state is mapped against some utility measure. A rational agent chooses the action that optimizes the expected utility of the outcome.

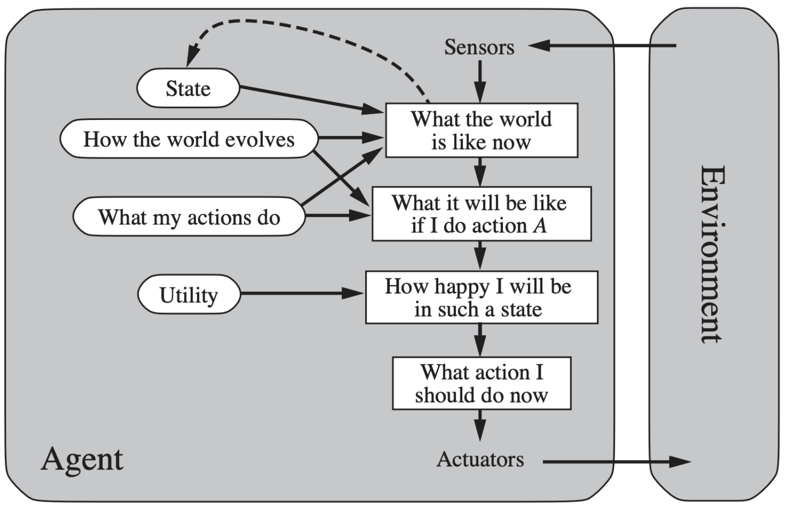
Sensors
State
What the world is like now
How the world evolves
What my actions do
What it will be like if I do action A
Utility
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

Figure 5: An utility-based agent.

4 Submission

1. Download the given `vacuum_cleaner.py` file that contains the complete codebase to do the lab.
2. Model the random agent program and simple reflex agent program to simulate the vacuum cleaner agent for the given example in Figure 1 during the lab. Then, submit your model (i.e., .py file). The file name should be in the following pattern: E16XXXLab1.py, where XXX is your registration number.

5 Important

- You are not supposed to use any Python libraries specifically design for agent modeling.
- We mark the final submission individually, and under no circumstance, you should copy somebody else's code.
- Copying someone else's code (including your group mate's) or showing your source code to anyone else will earn you zero mark for the whole lab exercise.

6 Deadline

The deadline for the submission is **2nd December 2022 (Friday) at 3.00 pm**.

7 Things You May Try

1. Model an agent program that can remove randomly distributed dirt in a rectangular world (i.e., 2D space) of a given size. When the world is cleaned, the agent should return to the start position and shut down.
2. Design a GUI to demonstrate the simulation in the 2D vacuum world.

8 Appendix

```
# This code is adopted from github.com/aimacode/aima-python repository
from statistics import mean
import random
import copy
import collections


class Thing(object):

    """This represents any physical entity that can appear in an Environment.
    Each thing can have a
    .__name__  slot (used for output only)."""

    def __repr__(self):
        return f"<{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 be overridden."""


class Agent(Thing):
    """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:
            def program(percept):
                return eval(input(f"Percept={percept}; action? "))
        assert isinstance(program, collections.Callable)
        self.program = program

    def can_grab(self, thing):
        """Returns True if this agent can grab this thing.
        Override for appropriate subclasses of Agent and Thing."""
        return False


def TableDrivenAgentProgram(table):
    """This agent selects an action based on the percept sequence.
    It is practical only for tiny domains.
    To customize it, provide as table a dictionary of all
    {percept_sequence:action} pairs."""
    percepts = []
    def program(percept):
        percepts.append(percept)
        action = table.get(tuple(percepts))
        return action
    return program


def RandomAgentProgram(actions):
    """An agent that chooses an action at random, ignoring all percepts."""
    return lambda percept: random.choice(actions)


def SimpleReflexAgentProgram(rules, interpret_input):
    """This agent takes action based solely on the percept."""
    def program(percept):
        state = interpret_input(percept)
        rule = rule_match(state, rules)
        action = rule.action
        return action
    return program


def ModelBasedReflexAgentProgram(rules, update_state):
    """This agent takes action based on the percept and state."""
    def program(percept):
        program.state = update_state(program.state, program.action, percept)
        rule = rule_match(program.state, rules)
        action = rule.action
        return action
    program.state = program.action = None
    return program


def rule_match(state, rules):
    """Find the first rule that matches state."""
    for rule in rules:
        if rule.matches(state):
            return rule


# Define three locations in the Vacuum world
loc_A, loc_B, loc_C = (0, 0), (1, 0), (2, 0)


def RandomVacuumAgent():
    return Agent("""Choose one of the actions from the vacuum environment randomly.""")


def TableDrivenVacuumAgent():
    table = {((loc_A, "Clean"),): "Right",
             ((loc_A, "Dirty"),): "Suck",
             ((loc_B, "Clean"),): "Right",
             ((loc_B, "Clean"),): "Left",
             ((loc_B, "Dirty"),): "Suck",
             # ...
             ((loc_A, "Clean"), (loc_A, "Clean")): "Right",
             ((loc_A, "Clean"), (loc_A, "Dirty")): "Suck",
             # ...
             ((loc_A, "Clean"), (loc_A, "Clean"), (loc_A, "Clean")): "Right",
             ((loc_A, "Clean"), (loc_A, "Clean"), (loc_A, "Dirty")): "Suck",
             # ...
             }
    return Agent(TableDrivenAgentProgram(table))


def ReflexVacuumAgent():
    """A simple reflex agent for the three-state vacuum environment."""
    def program(percept):
        location, status = percept
        if status == "Dirty":
            print(percept, "Suck")
            return "Suck"
        elif location == loc_A:
            print(percept, "Right")
            return "Right"
        # ...
        """Augment to simulate the three location problem environment (Figure 1)"""
    return Agent(program)


def AllSeeingReflexVaccumAgent():
    """More powerful sensors"""
    def program(percept):
        # Now reveals all locations and their statuses to agent(s)
        location, status = percept
        if status[loc_A] == status[loc_B] == status[loc_C] == "Clean":
            print(percept, "NoOp")
            return "NoOp"
        elif status[location] == "Dirty":
            print(percept, "Suck")
            return "Suck"
        elif location == loc_A and (status[loc_B] == "Dirty" or status[loc_C] == "Dirty"):
            print(percept, "Right")
            return "Right"
        elif location == loc_B and status[loc_A] == "Dirty":
            print(percept, "Left")
            return "Left"
        elif location == loc_B and status[loc_C] == "Dirty":
            print(percept, "Right")
            return "Right"
        # ...
    return Agent(program)


def BasicVaccumAgent():
    # Cannot see location, cannot store status, without randomizer
    def program(percept):
        if percept == "Dirty":
            print(percept, "Suck")
            return "Suck"
        else:
            # Decide random direction without randomizer inclusion.
            print(percept, "Right")
            return "Right"
    return Agent(program)


def SemiBasicVaccumAgent():
    # Cannot see location, cannot store status, with randomizer
    def program(percept):
        if percept == "Dirty":
            print(percept, "Suck")
            return "Suck"
        else:
            choice = random.randint(0, 1)
            if choice == 0:
                print(percept, "Left")
                return "Left"
            else:
                print(percept, "Right")
                return "Right"
    return Agent(program)


def ModelBasedVacuumAgent():
    """An agent that keeps track of what locations are clean or dirty."""
    model = {loc_A: None, loc_B: None, loc_C: None}


    def program(percept):
        """Same as ReflexVacuumAgent, except if everything is clean, do NoOp."""
        # Augmented to include the 3-room format
        location, status = percept
        model[location] = status  # Update the model here
        if model[loc_A] == model[loc_B] == model[loc_C] == "Clean":
            print(percept, "NoOp")
            return "NoOp"
        elif status == "Dirty":
            print(percept, "Suck")
            return "Suck"
        elif location == loc_A and status == "Clean":
            print(percept, "Right")
            return "Right"
        elif location == loc_B and status == "Clean":
            if model[loc_A] == "Clean":
                print(percept, "Right")
                return "Right"
            elif model[loc_C] == "Clean":
                print(percept, "Left")
                return "Left"
            else:
                position = random.randint(0, 1)
                if position == 0:
                    print(percept, "Left")
                    return "Left"
                elif position == 1:
                    print(percept, "Right")
                    return "Right"
        elif location == loc_C and status == "Clean":
            print(percept, "Left")
            return "Left"
        elif location == loc_A and (model[loc_B] or model[loc_C] == "Dirty"):
            print(percept, "Right")
            return "Right"
        elif location == loc_B and (model[loc_A] or model[loc_C] == "Dirty"):
            if model[loc_A] == "Dirty":
                print(percept, "Left")
                return "Left"
            elif model[loc_C] == "Dirty":
                print(percept, "Right")
                return "Right"
        elif location == loc_C and (model[loc_A] or model[loc_B] == "Dirty"):
            print(percept, "Left")
            return "Left"
    return Agent(program)


class Dirt(Thing):
    pass


class Environment(object):
    """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."""
        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."
        return [thing for thing in self.things
                if 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)
        assert thing not in self.things, "Don't add the same thing twice"
        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(f"  Thing to be removed: {thing} at {thing.location}")
            print(f"  from list: {[(thing, thing.location) for thing in self.things]}")
        if thing in self.agents:
            self.agents.remove(thing)


class TrivialVacuumEnvironment(Environment):
    """This environment has three locations, A, B, and C. Each can be Dirty
    or Clean.  The agent perceives its location and the location's
    status. This serves as an example of how to implement a simple
    Environment."""

    def __init__(self):
        super(TrivialVacuumEnvironment, self).__init__()
        self.status = {loc_A: random.choice(["Clean", "Dirty"]),
                       loc_B: random.choice(["Clean", "Dirty"]),
                       loc_C: random.choice(["Clean", "Dirty"])}
        print("The environment for this attempt is:")
        print(self.status)

    def thing_classes(self):
        return [Dirt, ReflexVacuumAgent, RandomVacuumAgent,
                TableDrivenVacuumAgent, ModelBasedVacuumAgent]

    def percept(self, agent):
        "Returns the agent's location, and the location status (Dirty/Clean)."
        return (agent.location, self.status[agent.location])

    def execute_action(self, agent, action):
        """Change agent's location and/or location's status; track performance.
        Score 10 for each dirt cleaned; -1 for each move."""
        if action == "Right":
            if agent.location == loc_B:
                agent.location = loc_C
            elif agent.location == loc_A:
                agent.location = loc_B
            agent.performance -= 1
        elif action == "Left":
            if agent.location == loc_B:
                agent.location = loc_A
            elif agent.location == loc_C:
                agent.location = loc_B
            agent.performance -= 1
        elif action == "Suck":
            if self.status[agent.location] == "Dirty":
                agent.performance += 10
            self.status[agent.location] = "Clean"
        for floorStatus in self.status:
            if floorStatus == "Dirty":
                agent.performance -= 2

    def default_location(self, thing):
        "Agents start in a location at random."
        return random.choice([loc_A, loc_B, loc_C])


class AllSeeingTrivialVaccumEnvironment(TrivialVacuumEnvironment):
    # Same as TrivialVaccumEnviornment, except gives all locations as well as statuses

    def percept(self, agent):
        "Returns the agent's location, and the location status (Dirty/Clean)."
        return (agent.location, self.status)


class BasicVaccumEnvironment(TrivialVacuumEnvironment):
    # Same as TrivialVaccumEnvironment; agent knows status yet no location

    def percept(self, agent):
        return (self.status[agent.location])


def run_times(EnvFactory, AgentFactory, steps, runs):
    for i in range(runs):
        env = EnvFactory()
        agent = AgentFactory()
        env.add_thing(agent)
        env.run(steps)
        score = str(agent.performance)
        print(score)


"""Test Random Agent Program"""
"""Test Simple Reflex Agent Program"""
```

***** End of Labsheet *****