CMSC 373: Artificial Intelligence
Lab#1/Assignment#1: Designing Pac-Man Agents

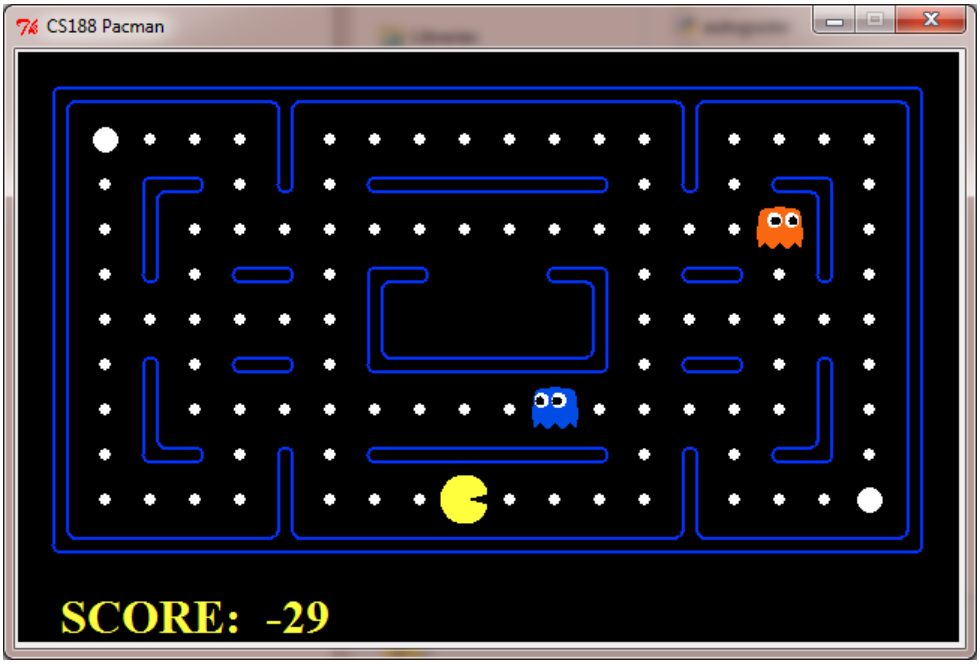

Figure 1: The Pac-Man World

Introduction

In this project, you will familiarize yourself with the Pac-Man World. Over the next few assignments your Pac-Man agent will find paths through its maze world, both to reach a particular location and to collect food efficiently. You will also build general search algorithms and apply them to Pac-Man scenarios.

The codebase for this project consists of several Python files, some of which you will need to read and understand in order to complete assignments, and some of which you can ignore. The Pac-Man code was developed by John DeNero and Dan Klein at UC Berkeley for the class CS188.

Before you begin, make sure that you have Python 3.X available on the computer/account you will be using. It is installed on all CS Lab computers (for Windows and Linux). You can check the version of Python by entering the command:

```
> python --version
Python 3.8.10
```

Commands you enter will be shown in red, followed by the system response/output in blue. You will enter these commands in a command shell after going to the code directory where you downloaded the Pac-Man code files (see below).

Step 1: Download Code

You can download the code archive by clicking this link:
http://cs.brynmawr.edu/Courses/cs373/fall2021/Code/search.zip

Extract the files into a directory/folder on your computer (or in your Linux account). A folder called, search will be created and in it you will find several dozen files. To ensure that you have a working version of the files, run the following command:

```
python pacman.py
```

You should see a game screen pop up (see Figure 1). This is a basic Pac-Man game. In the game, you control the movements of Pac-Man using arrow keys on your keyboard. Go ahead and try it.

The Pac-Man world is laid out as corridors (with shiny blue walls) where Pac-Man can move about. Little white pellets are sometimes littered throughout the corridors. This is food for Pac-Man (larger pellets are power food or capsules, try and figure out what those are for). In the world shown in Figure 1, Pac-Man has adversaries: colored ghosts that eat Pac-Man when it runs into them. Ghosts move about without eating any food. When Pac-Man is eaten, it dies and the game ends. The screen will disappear.

Step 2: Pac-Man Agent

In this and the next few assignments, you will be writing agent programs to control the actions of Pac-Man. That is, creating a Pac-Man agent. The code enables you to use different environments to try out your Pac-Man agent programs. To specify an environment (for example, **`testMaze`**), you use the command:

```
python pacman.py --layout testMaze
```

Go ahead and try it. It is a simple maze with one corridor. Here is one you will use more often:

```
python pacman.py --layout tinyMaze
```

Figure 2: Pac-Man agent in `tinyMaze`.

There are several other environments defined: **`mediumMaze`**, **`bigMaze`**, **`openSearch`**, etc. You can also vary the scale of the screen by using the **–zoom** option as shown below:

```
python pacman.py --layout tinyMaze --zoom 2
python pacman.py --layout bigMaze --zoom 0.5
```

All of these are single agent environments, the agent being Pac-Man. In these environments, Pan-Man always starts at the top right corner and, at the bottom left corner is a single food pellet (see picture above). The game ends when Pac-Man eats very last pellet (there can be pellets anywhere in its world).

Step 3: Learning the Pac-Man Grid and Actions

Grid: The environment is essentially a grid of squares. At any given time, Pac-Man occupies a square and faces one of the four directions: North, South, East, or West. There may be walls in between the square (like the t-shaped wall in **`tinyMaze`**) or entire squares might be blocked by walls (like the bottom right corner of **`tinyMaze`**. regardless, the location of Pac-Man is determined by the x- and y- coordinates of the grid (as shown below):

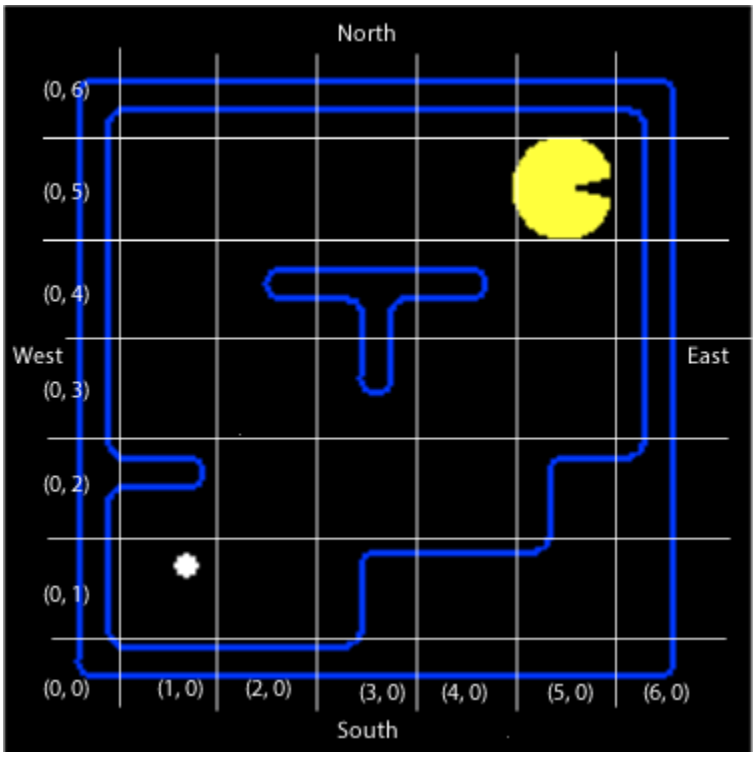

Figure 3: The Pac-Man Grid. Pac-Man is at position (5, 5). Food pellet is at (1, 1)

Actions

Pac-Man can only carry out the following actions:

- 'North': go 1 step north
- 'South': go one step south
- 'East': go one step east
- 'West': go one step west
- 'Stop': stop, do not move

Below, you will see how these are specified to be carried out.

Step 4: Diving into Some Code

Now that you are familiar with the basic world, it is time to get familiar with some of the code. Start by looking at the contents of the file **game.py**.

It defines several classes. In the code, you will see clearly marked sections:

```
######################
# Parts worth reading #
######################
```

And…

```
####################################
# Parts you shouldn't have to read #
####################################
```

Skim through the parts worth reading section of the code. Focus first on the following classes: **Agent**, **Directions**, and **Configuration**.

Agent

The **Agent** class is very simple. It is the class you will subclass to create your Pac-Man agent. For example, here is a very simple, and dumb, agent:

```
from game import Agent
from game import Directions

class DumbAgent(Agent):
    "An agent that goes West until it can't."

    def getAction(self, state):
        "The agent always goes West.”
        return Directions.WEST
```

The way it is set up, when you specify to the game (see below) that the Pac-Man will be controlled by an instance of a **DumbAgent**, the action returned by the **getAction()** method will be carried out at each time step. Important things to note in the above code are:

- You should create a new file called, **Agents.py**, in the same directory/folder as the rest of the code base. Enter the code above exactly as shown. Be sure to save the file.
- Every subclass of **Agent** (like **DumbAgent**) is required to implement a **getAction()** method. This is the method called in each time step of the game and as mentioned above, it should return a valid action for Pac-Man to carry out.
- Notice that we are importing the classes **Agent** and **Directions** from **game.py**

- The **`getAction()`** method is supplied a parameter: **`state`**, which it can use to find out about the current game state (more on this below). For now, we are ignoring it.
- Study the class **`Directions`** (defined in **`game.py`**).

Step 5: Run the code

Next run the Pac-Man game with its control as **`DumbAgent`** using the command:

```
python pacman.py --layout tinyMaze --pacman DumbAgent
```

The command above is specifying to run the Pac-Man game using the **`tinyMaze`** environment and the agent is controlled by the **`DumbAgent`**. What happens?

In the Pac-Man game, if the path to the grid is blocked and Pac-Man tries to go into it, the game crashes with an "**`Illegal action`**" exception. This is OK. After all, it is a dumb agent. We'll fix that next. Try the same agent in the **`mediumMaze`**. Same result, right? Good!

Step 6: Learning about GameState

Next, lets us try and use the information present in the state parameter. This is an object of type **`GameState`** which is defined in the file **`pacman.py`**. Study the **`GameState`** class closely and note the methods defined. Using these, you can get all kinds of information about the current state of the game. Then you can base your agent's action accordingly. Below, we show how you can use some of these and prevent the game from crashing.

```
from game import Agent
from game import Directions

class DumbAgent(Agent):
    "An agent that goes West until it can't."

def getAction(self, state):
        "The agent receives a GameState (defined in pacman.py)."
        print("Location: ", state.getPacmanPosition())
        print("Actions available: ", state.getLegalPacmanActions())
        if Directions.WEST in state.getLegalPacmanActions():
            print("Going West.")
            return Directions.WEST
        else:
            print("Stopping.")
            return Directions.STOP
```

As in Step 4, save this version of your program in **`Agents.py`** and run it on **`tinyMaze`**, as well as **`mediumMaze`**. Observe the behavior. Try out some of the other methods defined in **`GameState`** to get an idea of what information is available to your agent.

Step 7: A Random Agent

OK, now it is time to write your own agent code. It will be simple in what it does:

Based on the current options, pick a random action to carry out.

Code the above in a new class called, **`RandomAgent`** (in the **`Agents.py`** file). Run your agent in the **`tinyMaze`** environment as well as **`mediumMaze`** environment. Observe the agent's behavior. Does it get to the food? Always? Without crashing? Etc.

Step 8: Exploring Environments

See the files in the folder/directory **`layouts`**. Environments are specified using simple text files (**`*.lay`**) which are then rendered nicely by the graphics modules in the code base. Examine several layout files to see how to specify walls, ghosts, pacman, food, etc. Create a small environment of your own. Make sure it has walls and corridors, as well as some food. Save it as **`myLayout.lay`** in the **`layouts`** directory.

Run your **`RandomAgent`** in this environment and observe how it does.

Also, try your agent out in the **`openSearch`** environment (files are already provided in the layouts directory). Run your agent several times and record, on average, what score you get.

Step 9: A Better Random Agent

If you print out and look at the choice of actions at each step, you will notice that **`RandomAgent`** always includes a choice for the 'Stop' action. This tends to slow it down. Stopping is needed in situations where you need to evade ghosts. For now, in environments without any ghosts, you can choose not to pick the 'Stop' action.

Modify the **`RandomAgent`** code so that it never chooses 'Stop' as its action. Run the agent in **`openSearch`** and **`myLayout`** environments and observe how it does.

Step 10: Reflex Agents: Adding Percepts

What the Pac-Man agent can perceive is based on the methods of the **`GameState`** class which is defined in the file **`pacman.py`**. Open this file and let's look through the options.

Code Tip: The game has several different agents (Pac-Man and the ghosts). Each agent in the game has a unique **`index`**; Pac-Man is always **`index`** `0`, with ghosts starting at **`index`** `1`.

Pac-Man can perceive:

- Its position
- The position of all the ghosts
- The locations of the walls

- The positions of the capsules
- The positions of each food pellet
- The total number of food pellets still available
- Whether it has won or lost the game
- Its current score in the game

In addition, Pac-Man can also determine given the action it chooses what the next state of the environment will be, by using the method **`generatePacmanSuccessor()`**. It is clear from the methods available here that Pac-Man's environment is fully observable. Pac-Man's environment is also static because until it decides what to do and takes an action, the ghosts do not move (or, are not there).

In the file **`Agents.py`** create a new agent called **`ReflexAgent`**. This agent should look at the possible legal actions, and if one of these actions would cause a food pellet to be eaten, it should choose that action. If none of the immediate actions lead to food, it should choose randomly from the possibilities (excluding 'Stop'). Test your agent in both the **`openSearch`** and **`myLayout`** layouts.

```
python pacman.py --l openSearch --p ReflexAgent
```

What to hand in:

Your submission should include the following:

1. Answers to the following questions:
    a. Describe the behavior of **`RandomAgent`** from Step 7
    b. A screen shot of your **`myLayout`** environment from Step 8
    c. Describe the behavior of **`RandomAgent`** from Step 9
    d. Describe the behavior of **`ReflexAgent`** from Step 10
    e. For each of the percepts listed in Step 10, show what command/code enables you to access it. For example:
        Pac-man's postion: **`gameState.getPacmanPosition()`**
2. A short reflection from doing this exercise.
3. A printout of the file **`Agents.py`**
4. Staple these together, put your name on the front page and hand it in at the start of class on the due date.