Lab 5
Markov Decision Process

CSE 4712

ARTIFICIAL INTELLIGENCE LAB

NOVEMBER 3, 2022

Contents

1 Introduction

In this lab, you will implement value iteration. You will test your agents first on Gridworld (from class).

As in previous tasks, this lab includes an autograder for you to grade your answers on your machine. This can be run on all questions with the command:

```
python autograder.py
```

It can be run for one particular question, such as q2, by:

```
python autograder.py -q q2
```

It can be run for one particular test by commands of the form:

```
python autograder.py -t test_cases/q2/1-bridge-grid
```

The code for this lab contains the following files, available as `mdp.zip`.

Files you'll edit:	
`valueIterationAgents.py`	A value iteration agent for solving known MDPs.
`analysis.py`	A file to put your answers to questions given in the lab.
Files you might want to look at:	
`mdp.py`	Defines methods on general MDPs.
`learningAgents.py`	Defines the base classes `ValueEstimationAgent`, which your agents will extend.
`util.py`	Useful data structures for implementing MDPs.
`gridworld.py`	The Gridworld implementation.
Supporting files you can ignore:	
`environment.py`	Abstract class for general reinforcement learning environments. Used by `gridworld.py`.
`graphicsGridworldDisplay.py`	Gridworld graphical display.
`graphicsUtils.py`	Graphics utilities.
`textGridworldDisplay.py`	Plug-in for the Gridworld text interface.
`crawler.py`	The crawler code and test harness. You will run this but not edit it.
`graphicsCrawlerDisplay.py`	GUI for the crawler robot.
`qlearningAgents.py`	Q-learning agents for Gridworld, Crawler and Pacman.
`featureExtractors.py`	Classes for extracting features on (state, action) pairs. Used for the approximate Q-learning agent (in `qlearningAgents.py`).
`autograder.py`	Lab autograder
`testParser.py`	Parses autograder test and solution files
`testClasses.py`	General autograding test classes
`test_cases/`	Directory containing the test cases for each question
`reinforcementTestClasses.py`	Task 3 specific autograding test classes

Files to Edit: You will fill in portions of `valueIterationAgents.py` and `analysis.py` during the task. Please *do not* change the other files in this distribution or submit any of our original files other than this file.

Evaluation: Your code will be autograded for technical correctness. Please *do not* change the names

of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's judgments – will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; please don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. Office hours, Google Classroom, Emails are there for your support; please use them. We want these labs to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Google Classroom: Please be careful not to post spoilers.

2 MDPs

To get started, run Gridworld in manual control mode, which uses the arrow keys:

```
python gridworld.py -m
```

You will see the two-exit layout from class. The blue dot is the agent. Note that when you press up, the agent only actually moves north 80% of the time. Such is the life of a Gridworld agent!

You can control many aspects of the simulation. A full list of options is available by running:

```
python gridworld.py -h
```

The default agent moves randomly

```
python gridworld.py -g MazeGrid
```

You should see the random agent bounce around the grid until it happens upon an exit. Not the finest hour for an AI agent.

Note: The Gridworld MDP is such that you first must enter a pre-terminal state (the double boxes shown in the GUI) and then take the special 'exit' action before the episode actually ends (in the true terminal state called `TERMINAL_STATE`, which is not shown in the GUI). If you run an episode manually, your total return may be less than you expected, due to the discount rate (`-d` to change; 0.9 by default).

Look at the console output that accompanies the graphical output (or use `-t` for all text). You will be told about each transition the agent experiences (to turn this off, use `-q`).

As in Pacman, positions are represented by `(x, y)` Cartesian coordinates and any arrays are indexed by `[x][y]`, with '`north`' being the direction of increasing `y`, etc. By default, most transitions will receive a reward of zero, though you can change this with the living reward option (`-r`).

3 Question 1 (4 points): Value Iteration

Recall the value iteration state update equation:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V_k(s')\right]$$

Write a value iteration agent in `ValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`. Your value iteration agent is an offline planner that takes the number of iterations (option `-i`) as input in its initial planning phase. `ValueIterationAgent`

takes an MDP on construction and runs value iteration for the specified number of iterations before the constructor returns.

Value iteration computers k-step estimates of the optimal values, V_k. In addition to running value iteration, implement the following methods for `ValueIterationAgent` using V_k.

- `computeActionFromValues(state)` computes the best action according to the value function given by `self.values`.
- `computeQValueFromValues(state, action)` returns the Q-value of the (state, action) pair given by the value function given by `self.values`.

These quantities are all displayed in the GUI: values are numbers in squares, Q-values are numbers in square quarters, and policies are arrows out from each square.
Important: Use the "batch" version of value iteration where each vector V_k s computed from a fixed vector V_{k-1} (as shown in lecture), not the "online" version where one single weight vector is updated in place. This means that when a state's value is updated in iteration k based on the values of its successor states, the successor state values used in the value update computation should be those from iteration $k-1$ (even if some of the successor states had already been updated in iteration k). The difference is discussed in Sutton & Barto in Chapter 4.1 on page 91.
Note: A policy synthesized from values of depth k (which reflect the next k rewards) will actually reflect the next $k+1$ rewards (i.e. you return π_{k+1}). Similarly, the Q-values will also reflect one more reward than the values (i.e. you return Q_{k+1}).

You should return the synthesized policy π_{k+1}.
Hint: You may optionally use the `util.Counter` class in `util.py`, which is a dictionary with a default value of zero. However, be careful with `argMax`: the actual argmax you want may be a key, not in the counter!
Note: Make sure to handle the case when a state has no available actions in an MDP (think about what this means for future rewards).

To test your implementation, run the autograder:

```
python autograder.py -q q1
```

The following command loads your `ValueIterationAgent`, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state (`V(start)`, which you can read off of the GUI) and the empirical resulting average reward (printer after the 10 rounds of execution finish) are quite close.

```
python gridworld.py -a value -i 100 -k 10
```

Hint: On the default BookGrid, you can run value iteration for 5 iterations using the following command:

```
python gridworld.py -a value -i 5
```

The result of running the code should look like Figure 1.
Grading: Your value iteration agent will be graded in a new grid. We will check your values, Q-values, and policies after fixed numbers of iterations and at convergence (e.g. after 100 iterations).

4 Question 2 (1 point): Bridge Crossing Analysis

`BridgeGrid` is a grid world map with a low-reward terminal state and a high-reward terminal state separated by a narrow "bridge", on either side of which is a chasm of high negative reward. Here, the expected value of a state depends on a number of factors, including how the future rewards are discounted, how noisy the actions are, and how much reward is received on the non-terminal states. The agent starts near the low-reward state. With the default discount of 0.9 and the default noise

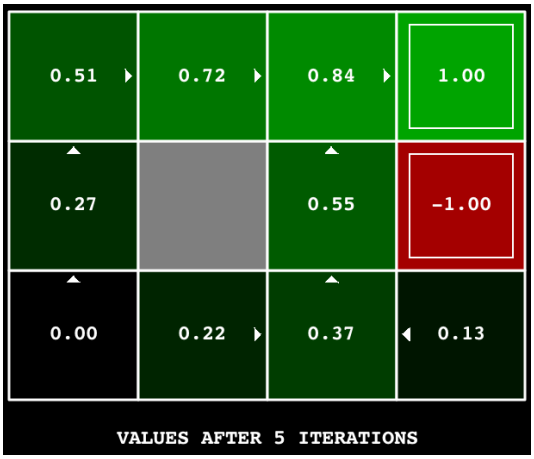

Figure 1: Result of running value iteration for 5 iterations in BookGrid

of 0.2, the optimal policy does not cross the bridge. Here, noise refers to how often an agent ends up in an unintended successor state when they perform an action. And discount determines how much the agent cares about rewards in the distant future relative to those in the immediate future. Change only ONE of the discount and noise parameters so that the optimal policy causes the agent to attempt to cross the bridge. Put your answer in `question2()` of `analysis.py`. (Noise refers to how often an agent) The default can be seen using the following command:

```
python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2
```

The resultant grid can be seen in Figure 2.

Figure 2: Default Values for Question 2

Grading: We will check that you only changed one of the given parameters, and that with this change, a correct value iteration agent should cross the bridge. To check your answer, run the autograder:

```
python autograder.py -q q2
```

5 Question 3 (5 points): Policies

Consider the `DiscountGrid` layout, shown in Figure 3. This grid has two terminal states with a positive payoff (in the middle row) – a close (near) exit with payoff +1, and a distant exit with payoff +10. The bottom row of the grid consists of terminal states with negative payoff (shown in red); each state in this "cliff" region has a payoff of -10. The starting state is the yellow square. We distinguish between two types of paths:

1. paths that "risk the cliff" and travel near the bottom row of the grid; these paths are shorter but risk earning a large negative payoff, and are represented by the red arrow in Figure 3.

2. paths that "avoid the cliff" and travel along the top edge of the grid. These paths are longer but are less likely to incur huge negative payoffs. These paths are represented by the green arrow in Figure 3.

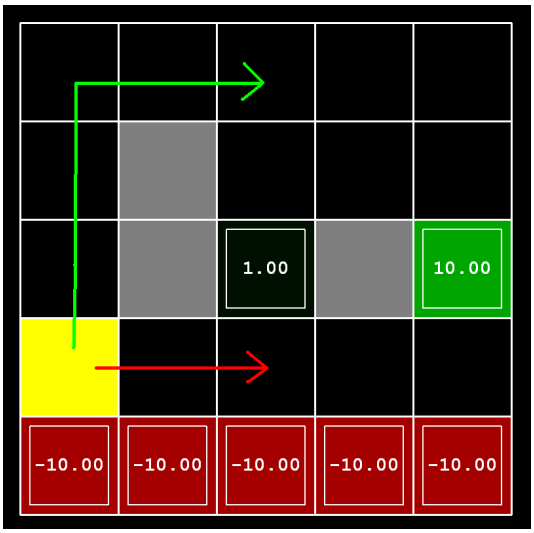

Figure 3: DiscountGrid Layout

In this question, you will choose values of the discount, noise, and living reward parameters for this MDP to produce optimal policies of several different types that are given below. **Your setting of the parameter values for each part should have the property that, if your agent followed its optimal policy without being subject to any noise, it would exhibit the given behavior.** If a particular behavior is not achieved for any setting of the parameters, assert that the policy is impossible by returning the string '`NOT POSSIBLE`'.

Here are the optimal policy types you should attempt to produce:

a. Prefer the close exit (+1), risking the cliff (-10)

b. Prefer the close exit (+1), avoiding the cliff (-1)

c. Prefer the distant exit (+10), risking the cliff (-10)

d. Prefer the distant exit (+10), avoiding the cliff (-10)

e. Avoid both exits and the cliff (so an episode should never terminate)

Write your answers in `analysis.py`: `question3a(a)` through `question3e()` should each return a 3-item tuple of discount, noise, and living reward. To check your answers, run the autograder:

```
python autograder.py -q q3
```

Note: You can check your policies in the GUI with commands like this:

```
python gridworld.py -a value -i 100 -g DiscountGrid -d 0.0 -n 0.0 -r 0.0
```

This will run the value iteration agent for 100 iterations on the `DiscountGrid` layout. The discount, noise and living reward all are set to 0.

As shown in Figure 4, using a correct answer to 3(a), the arrow in (0, 1) should point east, the arrow in (1, 1) should also point east, and the arrow in (2, 1) should point north.

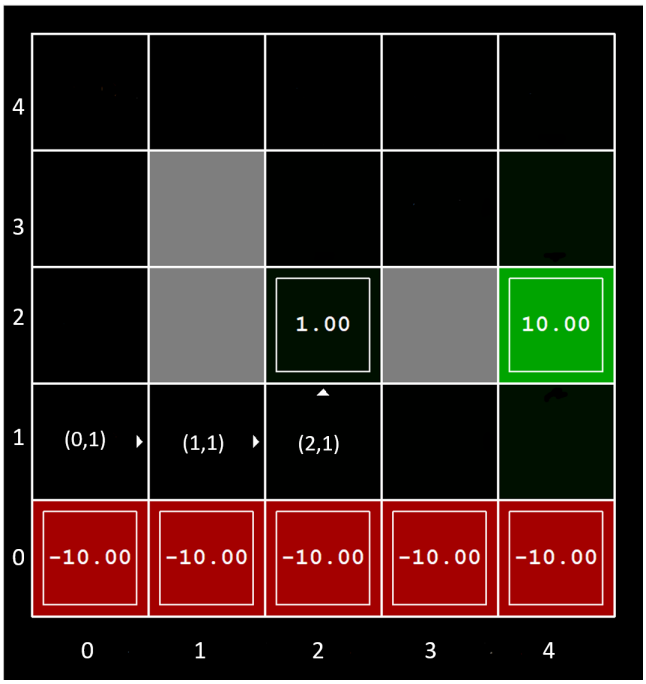

Figure 4: Sample policy with index shown in Parentheses

Note: On some machines you may not see an arrow. In this case, press a button on the keyboard to switch to qValues display, and mentally calculate the policy by taking the argmax of the available qValues for each state.
Grading: We will check that the desired policy is returned in each case.

6 Question 4 (1 point): Asynchronous Value Iteration

Write a value iteration agent in `AsynchronousValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`. Your value iteration agent is an offline planner that takes the number iterations (option `-i`) as input in its initial planning phase.
`AsynchronousValueIterationAgent` takes an MDP on construction and runs *cyclic* value iteration (described in the next paragraph) for the specified number of iterations before the constructor returns. Note that all this value iteration code should be placed inside the constructor (`__init__` method).

The reason this class is called `AsynchornousValueIterationAgent` is because we will update only **one** state in each iteration, as opposed to doing a batch-style update. Here is how cyclic value iteration works. In the first iteration, only update the value of the first state in the state's list. In the second iteration, only update the value of the second. Keep going until you have updated the value of each state once, then start back at the first state for the subsequent iteration. **If the state picked for updating is terminal, nothing happens in that iteration.** You can implement it as indexing into the states variable defined in the code skeleton.

As a reminder, here's the value iteration state update equation:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V_k(s')\right]$$

Value iteration iterates a fixed-point equation, as discussed in class. It is also possible to update the state values in different ways, such as in a random order (i.e., select a state randomly, update its value, and repeat) or in a batch style (as in Q1). In Q4, we will explore another technique.

`AsynchronousValueIterationAgent` inherits from `ValueIterationAgent` from Q1, so the only method you need to implement is `runValueIteration`. Since the superclass constructor calls `runValueIteration`, overriding it is sufficient to change the agent's behavior as desired.

Note: Make sure to handle the case when a state has no available actions in an MDP (think about what this means for future rewards).

To test your implementation, run the autograder. It should take less than a second to run. **If it takes much longer, you may run into issues later in the tasks, so make your implementation more efficient now.**

```
python autograder.py -q q4
```

The following command loads your `AsynchronousValueIterationAgent` in the Gridworld, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state (`V(start)`, which you can read off of the GUI) and the empirical resulting average reward (printed after the 10 rounds of execution finish) are quite close.

```
python gridworld.py -a asynchvalue -i 1000 -k 10
```

Grading: Your value iteration agent will be graded on a new grid. We will check your values, Q-values, and policies after fixed numbers of iterations and at convergence (e.g., after 1000 iterations).

7 Evaluation

Once you are done with the tasks, call your course teacher and show them the autograder results and codes.

8 Submission

Submit one file: `StudentID_L5.pdf` (`StudentID` will be replaced by your student ID) under **Assignment 5** on **Google Classroom**. The file can contain (but not limited to) your working code, analysis of the problem, explanation of the solutions, any interesting findings, any problems that you faced and how you solved it, behavior of the code for different hyperparameters, etc. All in all, the file is treated as lab report containing your **code** and **findings**.

You will have 2 weeks to submit the file.