Project 3: MDPs

Download the zip-file with the source-code here:

Introduction

New v2: fixed the qLearningAgent error! (unnecessary import)

mdps_v2.zip

In this project, you will implement value iteration for known MDPs. You will test your agents first on Gridworld (as seen in class).

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

```
python autograder.py
```

(Or, if you followed our Pycharm-specific setup, you can run the autograder by right-clicking the file `autograder.py` and clicking "run")

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-question-2.1
```

The code for this project contains the following files:

	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 project.
	Files you might want to look at:
mdp.py	Defines methods on general MDPs.

learningAgents.py	Defines the base classes ValueEstimationAgent, which your agent will extend.
util.py	Utilities, including util.Counter
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.
graphicsCrawlerDisplay.py	GUI for a crawler robot (not used).
autograder.py	Project 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	Project 3 specific autograding test classes

Files to Edit and Submit: You will fill in portions of valueIterationAgents.py, and analysis.py during the assignment. Once you have completed the assignment, you will submit these files to Toledo.

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 judgements – 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: On Toledo, you can also use the "Discussion board" feature to ask questions to you collegues. **Be careful not to post spoilers!**

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).

Question 1 (5 points): Value Iteration

Recall the value iteration state update equation:

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

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, not a reinforcement learning agent, and so the relevant training option is the number of iterations of value iteration it should run (option -i) 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 computes *k*-step estimates of the optimal values, V_k. In addition to runValueIteration, implement the following methods for ValueIterationAgent using V_k:

- computeActionFromValues(state) computes the best action according to the value function given by self.values.

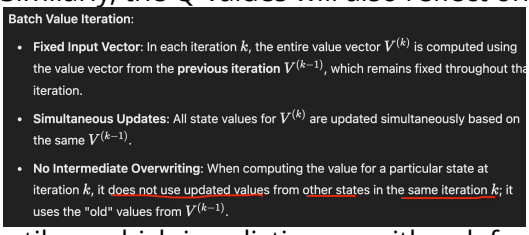
- computeQValueFromValues(state, action) returns the Q-value of the (state, action) pair given by the value function given by self.values.

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 is computed from a fixed vector V_{k-1} (like 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 (https://web.archive.org/web/20230417150626/https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf) 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
```

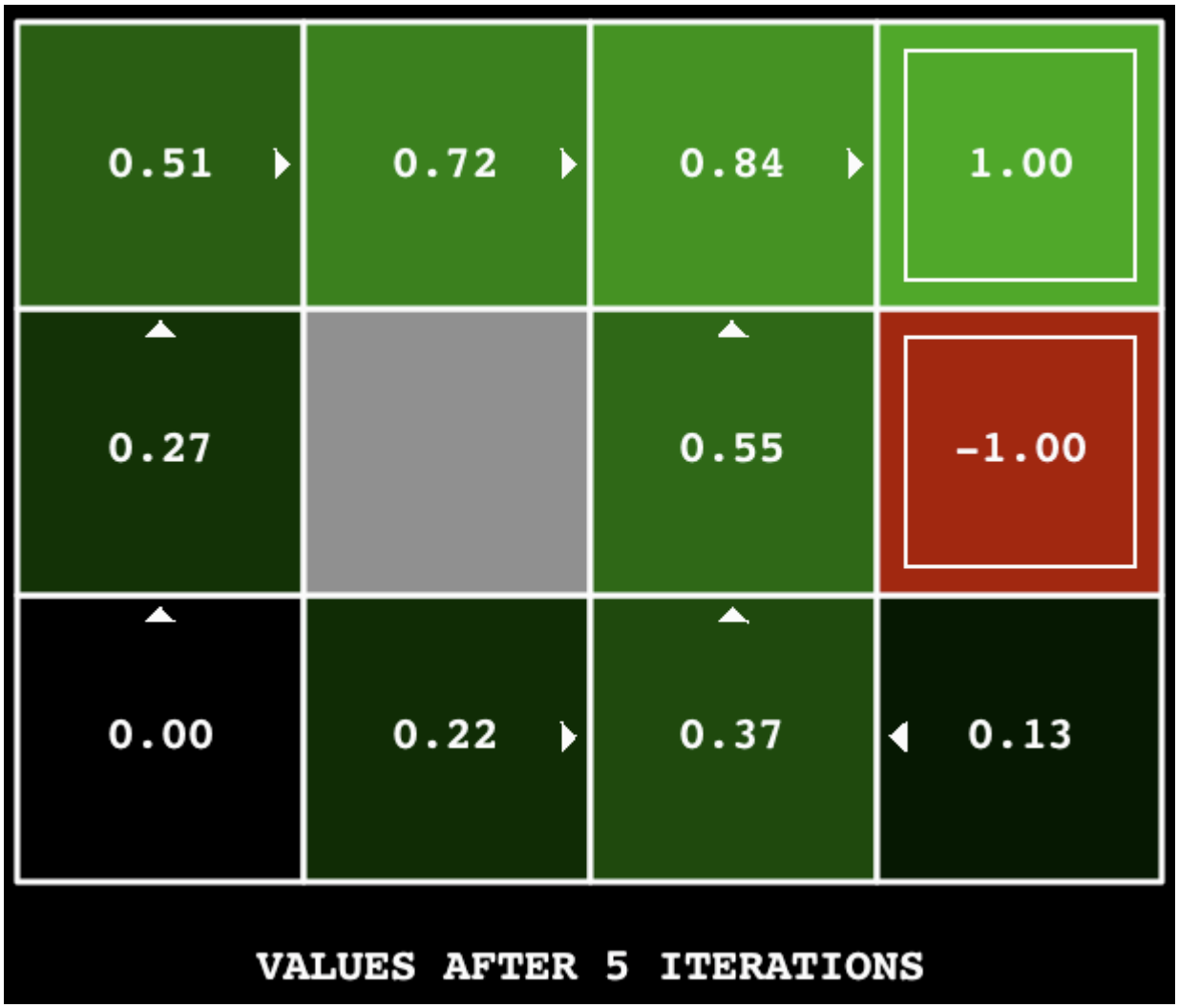
The following command loads your ValueIterationAgent, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state (V(start), which you can read off of the GUI) and the empirical resulting average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

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

Hint: On the default BookGrid, running value iteration for 5 iterations should give you this output:

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

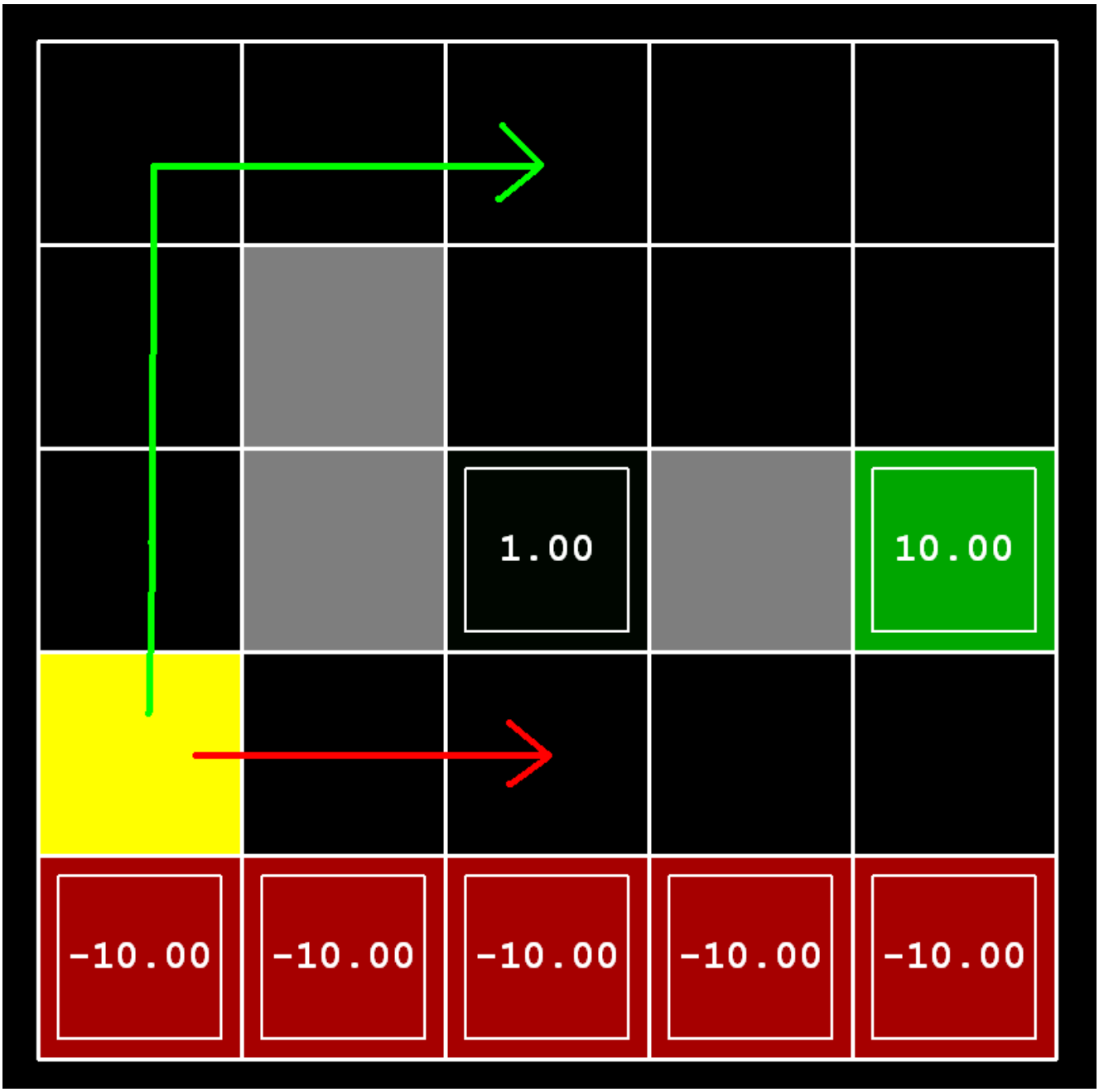
If your value iteration implementation is correct, the computed value $V(start)$ (theoretical prediction) should be close to the **empirical average reward** obtained by executing the policy in the environment.



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 100 iterations).

Question 2 (5 points): Policies

Consider the DiscountGrid layout, shown below. This grid has two terminal states with positive payoff (in the middle row), a close 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 payoff -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 the figure below. (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 the figure below.



In this question, you will choose settings of the discount, noise, and living reward parameters for this MDP to produce optimal policies of several different types. **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:

1. Prefer the close exit (+1), risking the cliff (-10)
2. Prefer the close exit (+1), but avoiding the cliff (-10)
3. Prefer the distant exit (+10), risking the cliff (-10)
4. Prefer the distant exit (+10), avoiding the cliff (-10)
5. Avoid both exits and the cliff (so an episode should never terminate)

To see what behavior a set of numbers ends up in, run the following command to see a GUI:

```
python gridworld.py -g DiscountGrid -a value --discount [YOUR_DISCOUNT] --
noise [YOUR_NOISE] --livingReward [YOUR_LIVING_REWARD]
```