CSCI 3202 Introduction to
Spring 2019 Artificial Intelligence

Assignment 3

MDPs and Reinforcement Learning

Aim: Implement MDP's and Reinforcement learning on your own

Issued: February 20th

Due: March 6th 4pm

Submission: Solutions should be submit via Moodle. The online submission will be available from Febraury 27th. Run the provided autograder locally **before** submitting to Moodle

Policy: Work must be done individually, refer to collaboration policy in syllabus for more details.

Introduction

In this project, you will implement value iteration and Q-learning. You will test your agents first on Gridworld (from class), then apply them to a simulated robot controller (Crawler) and Pacman.

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
```

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 project contains the following files, available as a zip archive.

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).

Q1. [4 pts] 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, 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 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 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).

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).

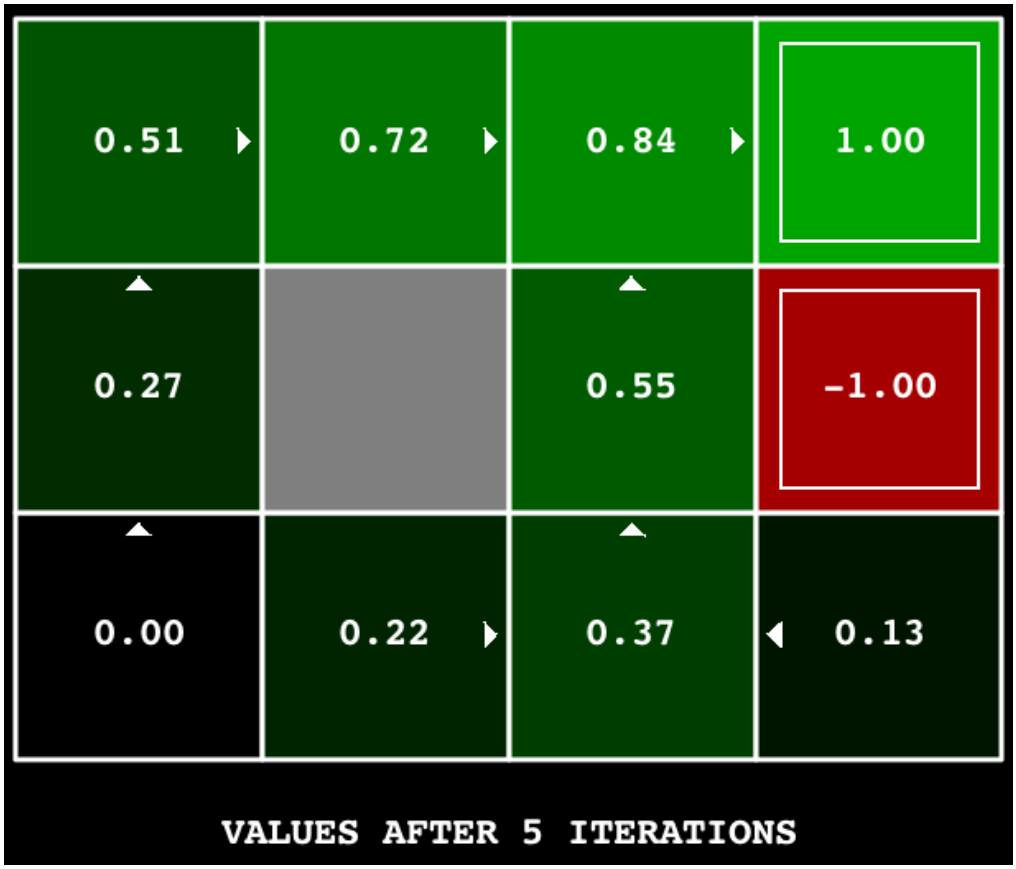
To test your implementation, run the autograder:

```
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`

Q2. [12 pts] Q-Learning

Note that your value iteration agent does not actually learn from experience. Rather, it ponders its MDP model to arrive at a complete policy before ever interacting with a real environment. When it does interact with the environment, it simply follows the precomputed policy (e.g. it becomes a reflex agent). This distinction may be subtle in a simulated environment like a Gridword, but it's very important in the real world, where the real MDP is not available.

You will now write a Q-learning agent, which does very little on construction, but instead learns by trial and error from interactions with the environment through its update(state, action, nextState, reward) method. A stub of a Q-learner is specified in `QLearningAgent` in `qlearningAgents.py`, and you can select it with the option '-a q'. For this question, you must implement the update, `computeValueFromQValues`, `getQValue`, and `computeActionFromQValues` methods.

Note: For `computeActionFromQValues`, you should break ties randomly for better behavior. The `random.choice()` function will help. In a particular state, actions that your agent hasn't seen before still have a Q-value, specifically a Q-value of zero, and if all of the actions that your agent has seen before have a negative Q-value, an unseen action may be optimal.

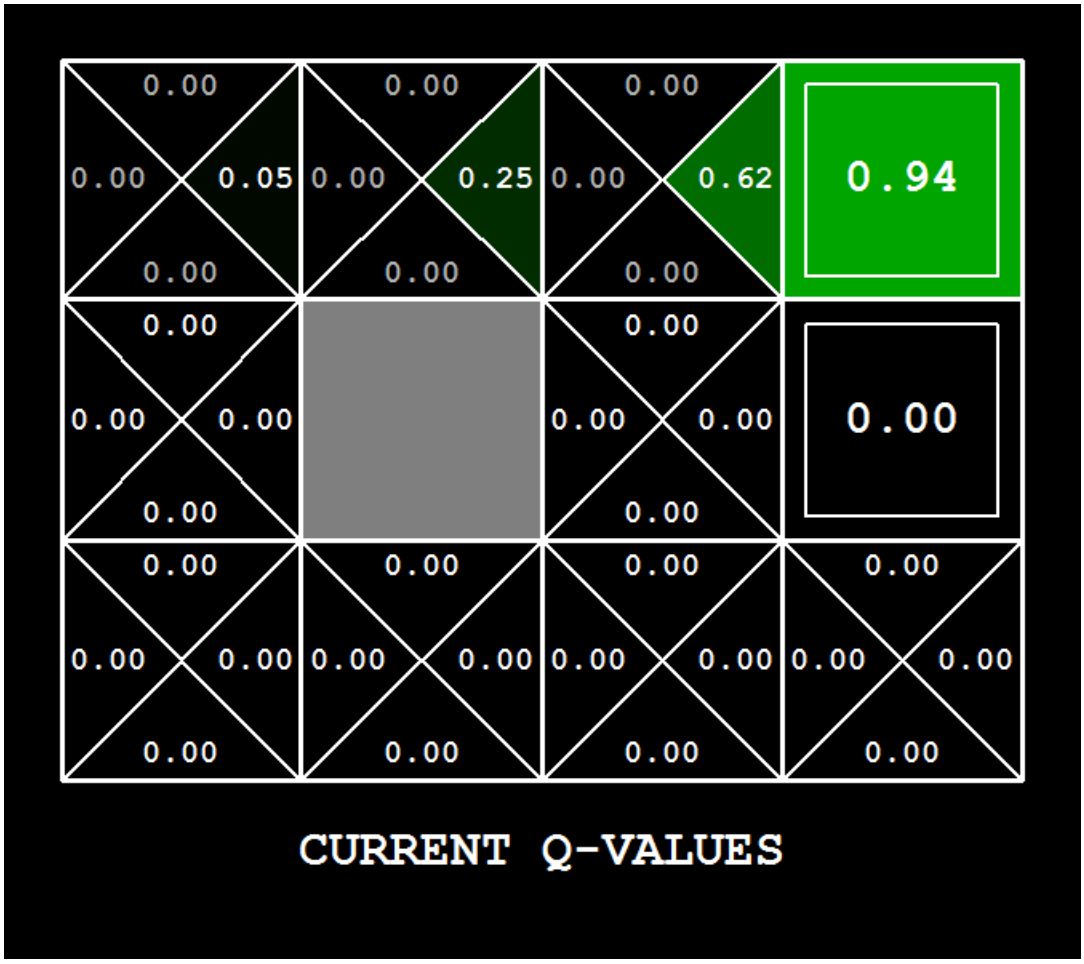
Important: Make sure that in your computeValueFromQValues and computeActionFromQValues functions, you only access Q values by calling `getQValue` .

With the Q-learning update in place, you can watch your Q-learner learn under manual control, using the keyboard:

`python gridworld.py -a q -k 5 -m`

Recall that `-k` will control the number of episodes your agent gets to learn. Watch how the agent learns about the state it was just in, not the one it moves to, and "leaves learning in its wake." Hint: to help with debugging, you can turn off noise by using the `--noise 0.0` parameter (though this obviously makes Q-learning less interesting). If you

manually steer Pacman north and then east along the optimal path for four episodes, you should see the following Q-values:



Grading: We will run your Q-learning agent and check that it learns the same Q-values and policy as our reference implementation when each is presented with the same set of examples. To grade your implementation, run the autograder:

```
python autograder.py -q q6
```