REINFORCEMENT LEARNING (PART 1)

Based on Pacmal Projects from CS 188, UC-Berkeley.
Prepared by Rubén Martínez Cantín rmcantin@unizar.es

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.

Files to Edit and Submit: In this session, you will fill in portions of `valueIterationAgents.py` and `analysis.py` during the assignment. In the next lab, you will also fill `qlearningAgents.py`, and complete the `analysis.py` file. Please *do not* change the other files in this distribution or submit any of our original files other than these files. You also need to submit a PDF file explaining each question. **This explanation is especially important for the analysis questions.**

Files you'll edit:	
`valueIterationAgents.py`	A value iteration agent for solving known MDPs.
`qlearningAgents.py`	Q-learning agents for Gridworld, Crawler and Pacman.
`analysis.py`	A file to put your answers to questions given in the project.
Files you should read but NOT edit:	
`mdp.py`	Defines methods on general MDPs.
`learningAgents.py`	Defines the base classes `ValueEstimationAgent` and `QLearningAgent`, which your agents will extend.
`util.py`	Utilities, including `util.Counter`, which is particularly useful for Q-learners.
`gridworld.py`	The Gridworld implementation.
`featureExtractors.py`	Classes for extracting features on (state, action) pairs. Used for the approximate Q-learning agent (in `qlearningAgents.py`).
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. You will run this but not edit it.
`graphicsCrawlerDisplay.py`	GUI for the crawler robot.
`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

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: Value Iteration

Recall the value iteration state update equation:

$$V_{k+1}(x) = \max_a \sum_{x' \in \mathcal{X}} p(x'|x,a) \left(R(x,a,x') + \gamma V_k(x') \right)$$

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 the 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 75.

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:

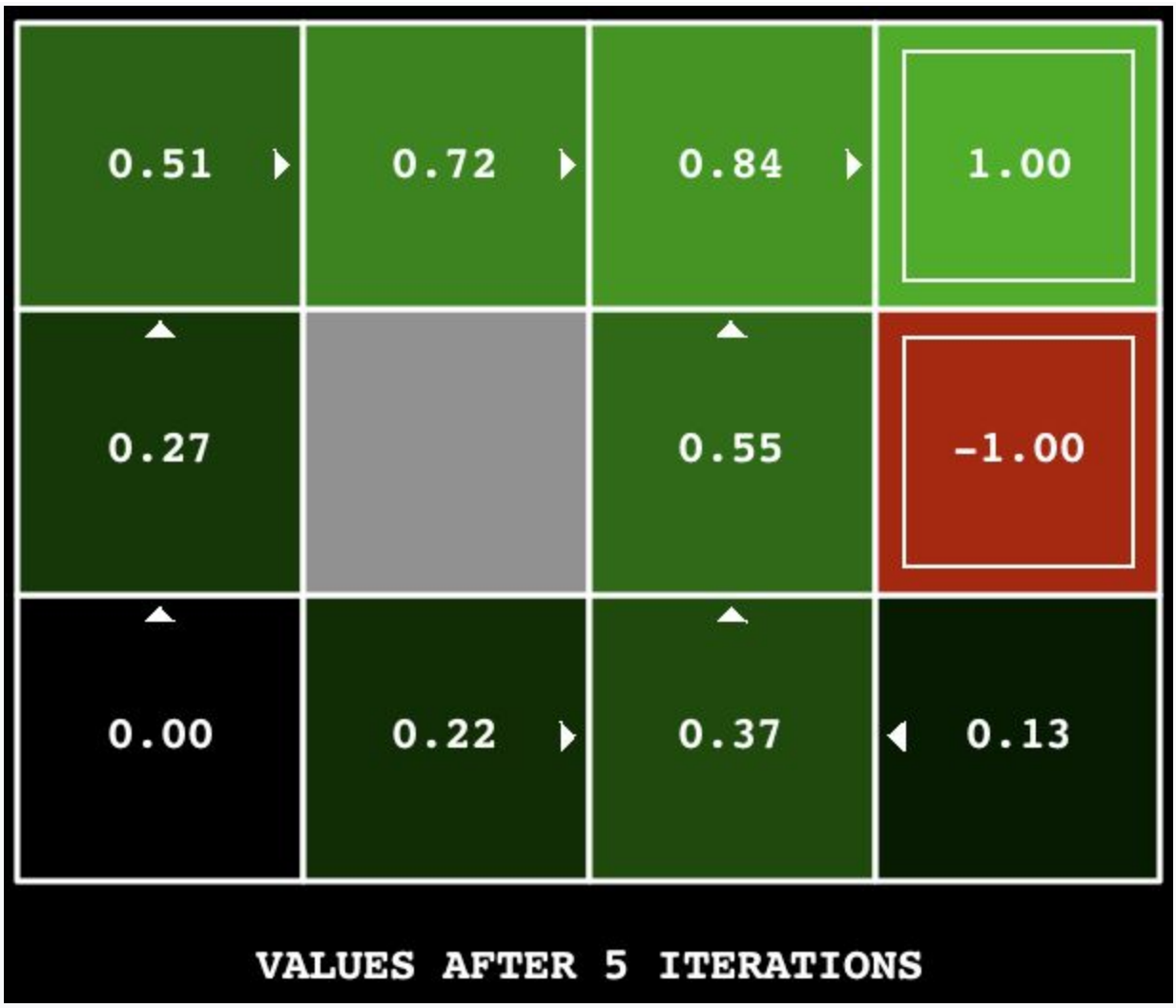
```
python autograder.py -q q1
```

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
```

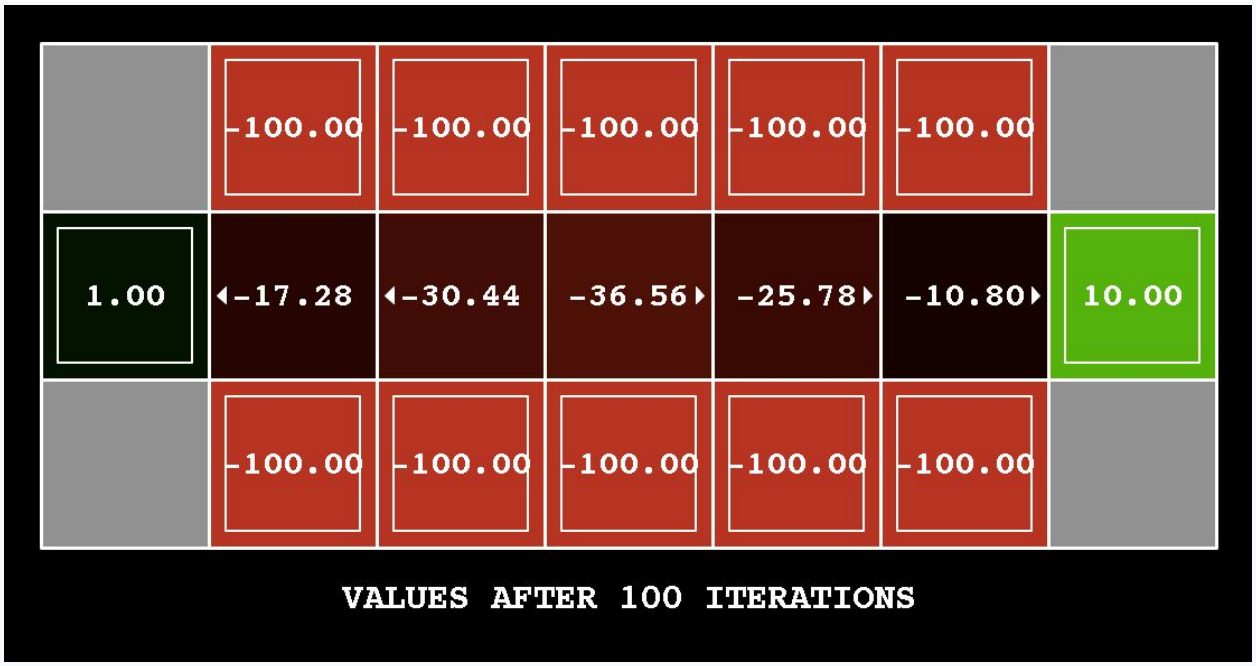


Grading: Your value iteration agent will be graded on a new grid. We will check your values, Q-values, and policies after fixed numbers of iterations and at convergence (e.g. after 100 iterations).

Question 2: 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. The agent starts near the low-reward state. With the default discount of 0.9 and the default noise of 0.2, the optimal policy does not cross the bridge. 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 ends up in an unintended successor state when they perform an action.) The default corresponds to:

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

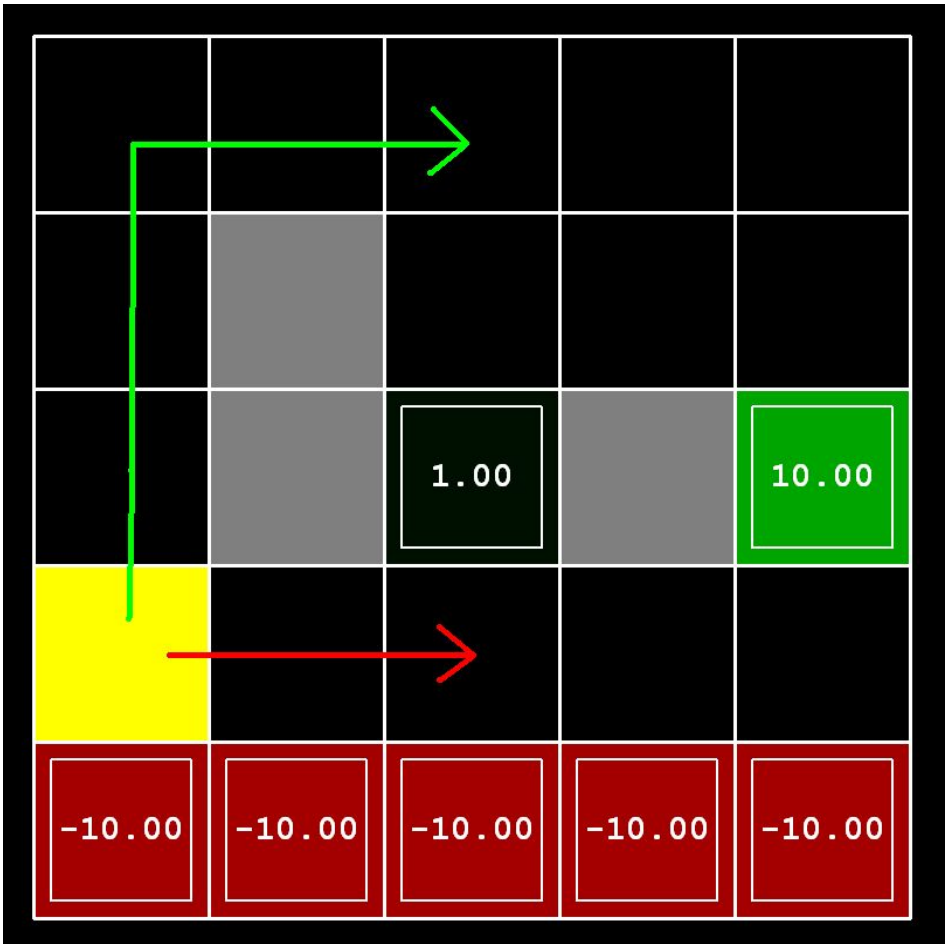


Grading: We will check that you only changed one of the given parameters, and that with this change, a correct value iteration agent should cross the bridge. To check your answer, run the autograder:

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

Question 3: 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 check your answers, run the autograder:

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

`question3a()` through `question3e()` should each return a 3-item tuple of (discount, noise, living reward) in `analysis.py`.

Note: You can check your policies in the GUI. For example, 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.

Note: On some machines you may not see an arrow. In this case, press a button on the keyboard to switch to qValue display, and mentally calculate the policy by taking the arg max of the available qValues for each state.

Grading: We will check that the desired policy is returned in each case.

Question 4: 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, 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. `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 `AsynchronousValueIterationAgent` 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 states 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:

$$V_{k+1}(x) = \max_a \sum_{x' \in \mathcal{X}} p(x'|x,a) \left(R(x,a,x') + \gamma V_k(x') \right)$$

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 this question, 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 project, 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).

Question 5: Prioritized Sweeping Value Iteration (Optional)

You will now implement `PrioritizedSweepingValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`. Note that this class derives from `AsynchronousValueIterationAgent`, so the only method that needs to change is `runValueIteration`, which actually runs the value iteration.

Prioritized sweeping attempts to focus updates of state values in ways that are likely to change the policy.

For this project, you will implement a simplified version of the standard prioritized sweeping algorithm, which is described in this paper. We've adapted this algorithm for our setting. First, we define the predecessors of a state `x` as all states that have a nonzero probability of reaching `x` by taking some action `a`. Also, `theta`, which is passed in as a parameter, will represent our tolerance for error when deciding whether to update the value of a state. Here's the algorithm you should follow in your implementation.

- Compute predecessors of all states.
- Initialize an empty priority queue.
- For each non-terminal state `x`, do: (note: to make the autograder work for this

question, you must iterate over states in the order returned by `self.mdp.getStates()`)
    - Find the absolute value of the difference between the current value of `x` in `self.values` and the highest Q-value across all possible actions from `x` (this represents what the value should be); call this number `diff`. Do NOT update `self.values[s]` in this step.
    - Push `x` into the priority queue with priority `-diff` (note that this is negative). We use a negative because the priority queue is a min heap, but we want to prioritize updating states that have a higher error.
- For `iteration` in `0, 1, 2, ..., self.iterations - 1`, do:
    - If the priority queue is empty, then terminate.
    - Pop a state `x` off the priority queue.
    - Update the value of `x` (if it is not a terminal state) in `self.values`.
    - For each predecessor `p` of `x`, do:
        - Find the absolute value of the difference between the current value of `p` in `self.values` and the highest Q-value across all possible actions from `p` (this represents what the value should be); call this number `diff`. Do NOT update `self.values[p]` in this step.
        - If `diff > theta`, push `p` into the priority queue with priority `-diff` (note that this is negative), as long as it does not already exist in the priority queue with equal or lower priority. As before, we use a negative because the priority queue is a min heap, but we want to prioritize updating states that have a higher error.

A couple of important notes on implementation:

- When you compute predecessors of a state, make sure to store them in a set, not a list, to avoid duplicates.
- Please use `util.PriorityQueue` in your implementation. The `update` method in this class will likely be useful; look at its documentation.

To test your implementation, run the autograder. It should take about 1 second to run. If it takes much longer, you may run into issues later in the project, so make your implementation more efficient now.

```
python autograder.py -q q5
```

You can run the `PrioritizedSweepingValueIterationAgen` in the Gridworld using the following command.

```
python gridworld.py -a priosweepvalue -i 1000
```

Grading: Your prioritized sweeping 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).