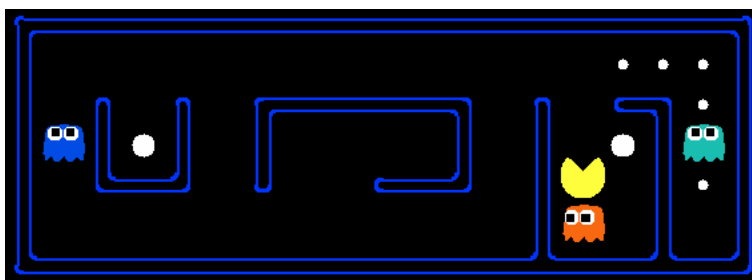


INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Exercise 6: Reinforcement Learning



Deadline: August 1st 2024

1 Introduction

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

The code for this project contains the following files, which are available in a [zip archive](#).

Files you'll edit:

[valueIterationAgents.py](#) A value iteration agent for solving known MDPs.

[qlearningAgents.py](#) Q-learning agents for Gridworld and Crawler.

[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 and test harness. You will run this but not edit it.

[graphicsCrawlerDisplay.py](#) GUI for the crawler robot.

What to submit: You will fill in portions of [valueIterationAgents.py](#), [qlearningAgents.py](#), and [analysis.py](#) during the assignment. You should submit only these files and a README.txt (case sensitive) as a tar file in the moodle website. Each team should submit exactly one file! Please don't change any other file.

Evaluation: Your code will be autograded for technical correctness. The autograder machine is running Python 3.7 (don't use Python 2.7). Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. Please make sure you follow the README format **exactly**.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy code from someone else 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: If you have any question, you are probably not alone. Please post your questions via the [relevant exercise forum](#) on the [course Moodle](#). **Please do not write to our personal e-mail addresses!**

README format: Please submit a README.txt file. The README should include the following lines (exactly):

1. id1 --- student 1 id
2. id2 --- student 2 id
3. ***** --- 5 stars denote end of i.d information
4. comments

For an example check out the [README.txt](#) provided with your project. This README will be read by a script, calling the autograder. Note that if you decide to submit alone, you should remove line 2, i.e.:

1. id1 --- student 1 id
2. ***** --- 5 stars denote end of i.d information
3. comments

2 MDPs

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

```
python3 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:

```
python3 gridworld.py -h
```

The default agent moves randomly:

```
python3 gridworld.py -g MazeGrid
```

You should see the random agent bounce around the grid until it happens upon an exit.

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 (use `-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 Pac-Man, 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 (6 points)

Write a value iteration agent in `ValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`. Your value iteration agent is an offline solver, 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 a MDP on construction and applies 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 .

- `getValue(state)` returns the value of a state.
- `getPolicy(state)` returns the best action according to computed values.
- `getQValue(state, action)` returns the Q-value of the `(state, action)` pair.

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 only a single weight vector is updated in place. The difference is discussed in [Sutton & Barto](#) in the 5th paragraph of chapter 4.1.

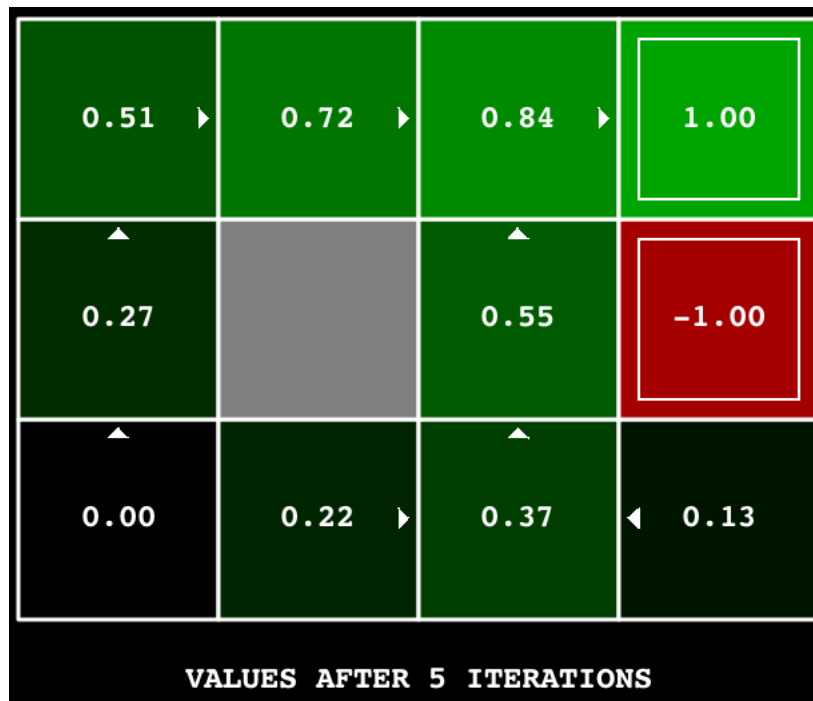
Note: A policy synthesized from values of depth k (which reflect the next k rewards) will actually reflect the next $k + 1$ rewards (i.e. you return π_{k+1}). Similarly, the Q-values will also reflect one more reward than the values (i.e. you return Q_{k+1}). You may assume that 100 iterations is enough for convergence in the questions below.

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)`) and the empirical resulting average reward are quite close.

```
python3 gridworld.py -a value -i 100 -k 10
```

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

```
python3 gridworld.py -a value -i 5
```



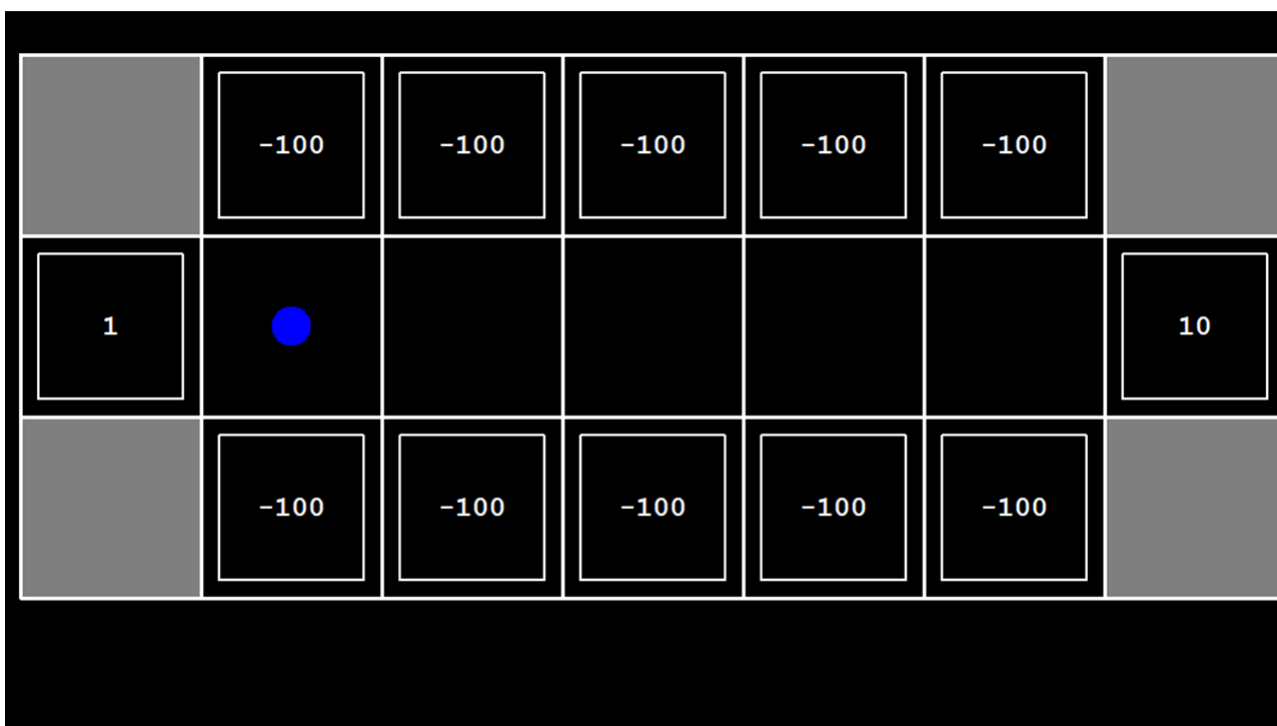
Your value iteration agent will be graded on a new grid. We will check your values, Q-values, and policies after fixed numbers of iterations and at convergence (e.g. after 100 iterations).

Hint: Use the `util.Counter` class in `util.py`, which is a dictionary with a default value of zero. Methods such as `totalCount` should simplify your code. However, be careful with `argMax`: the actual `argmax` you want may not be a key in the counter!

Question 2 (1 point)

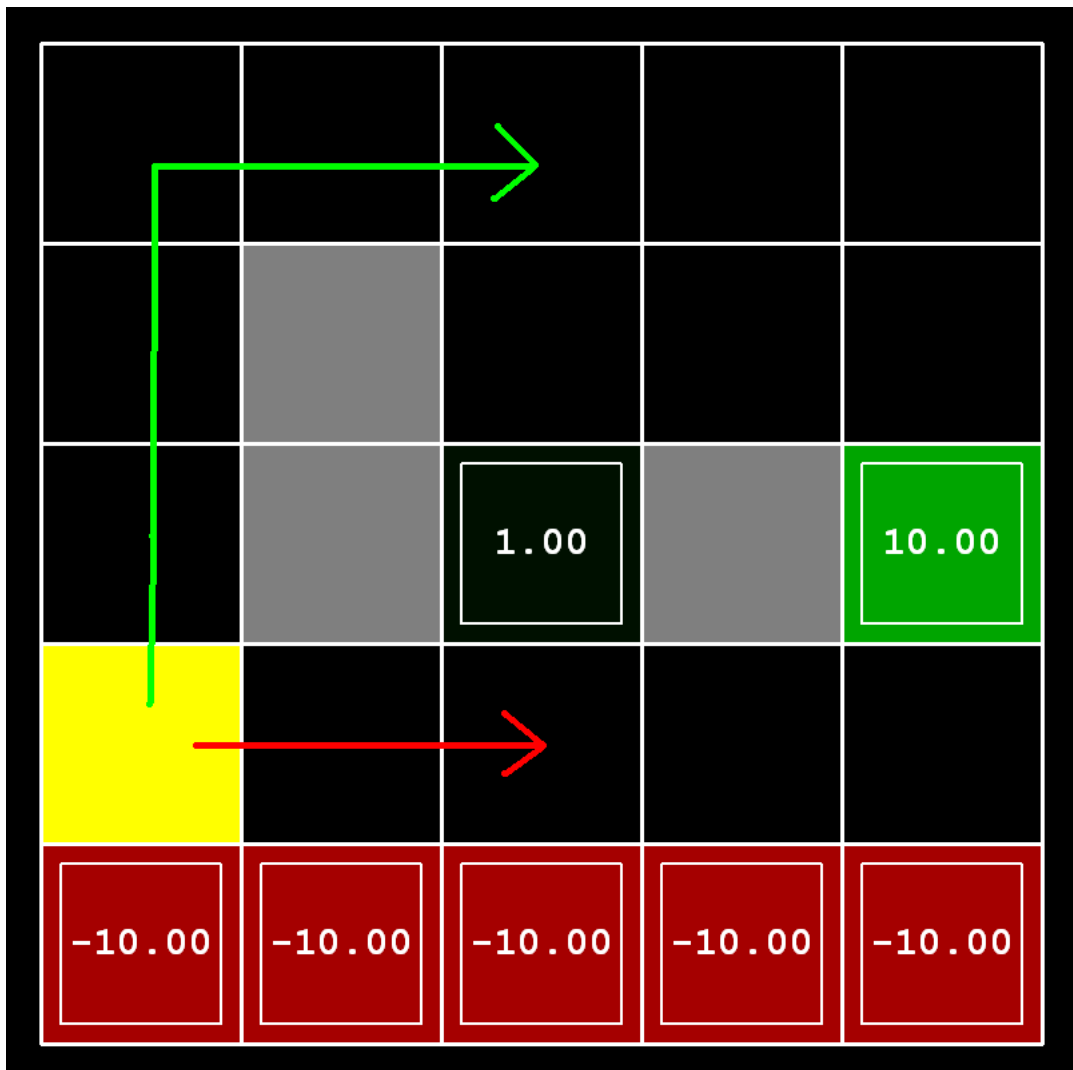
On `BridgeGrid` (shown bellow) with the default discount of 0.9 and the default noise of 0.2, the optimal policy does not cross the bridge. Change only ONE of the discount and noise parameters so that the optimal policy causes the agent to attempt to cross the bridge. Put your answer in `question2()` 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:

```
python3 gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2
```



Question 3 (5 points)

Consider the `DiscountGrid` layout, shown below. This grid has two terminal states with positive payoff (shown in green), 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.



Give an assignment of parameter values for `discount`, `noise`, and `livingReward` which produce the following optimal policy types or state that the policy is impossible by returning the string 'NOT POSSIBLE'. The default corresponds to:

```
python3 gridworld.py -a value -i 100 -g DiscountGrid --discount 0.9 --noise 0.2 --livingReward 0.0
```

- (a) Prefer the close exit (+1), risking the cliff (-10)
- (b) Prefer the close exit (+1), but avoiding the cliff (-10)
- (c) Prefer the distant exit (+10), risking the cliff (-10)
- (d) Prefer the distant exit (+10), avoiding the cliff (-10)
- (e) Avoid both exits (also avoiding the cliff)

`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.

3 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 `Gridworld`, but it's very important in the real world, where the real MDP is not available.

Question 4 (5 points)

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`, `getValue`, `getQValue` and `getPolicy` methods.

Note: For `getValue` and `getPolicy`, 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 you only access Q-values by calling `getQValue` in your `getValue`, `getPolicy` functions. This abstraction will be useful for question 9 when you override `getQValue` to use features of state-action pairs rather than state-action pairs directly.

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

```
python3 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."

Question 5 (2 points)

Complete your Q-learning agent by implementing epsilon-greedy action selection in `getAction`, meaning it chooses random actions epsilon of the time, and follows its current best Q-values otherwise.

```
python3 gridworld.py -a q -k 100
```

Your final Q-values should resemble those of your value iteration agent, especially along well-traveled paths. However, your average returns will be lower than the Q-values predict because of the random actions and the initial learning phase.

You can choose an element from a list uniformly at random by calling the `random.choice` function. You can simulate a binary variable with probability p of success by using `util.flipCoin(p)`, which returns `True` with probability p and `False` with probability $1 - p$.

Question 6 (1 points)

First, train a completely random Q-learner with the default learning rate on the noiseless `BridgeGrid` for 50 episodes and observe whether it finds the optimal policy.

```
python3 gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1
```

Now try the same experiment with an epsilon of 0. Is there an epsilon and a learning rate for which it is highly likely (greater than 99%) that the optimal policy will be learned after 50 iterations? `question6()` should return EITHER a 2-item tuple of (epsilon, learning rate) OR the string 'NOT POSSIBLE' if there is none. Epsilon is controlled by `-e`, learning rate by `-l`.

Question 7 (1 point)

With no additional code, you should now be able to run a Q-learning crawler robot:

```
python3 crawler.py
```

If this doesn't work, you've probably written some code too specific to the **GridWorld** problem and you should make it more general to all MDPs. You will receive full credit if the command above works without exceptions.

This will invoke the crawling robot from class using your Q-learner. Play around with the various learning parameters to see how they affect the agent's policies and actions. Note that the step delay is a parameter of the simulation, whereas the learning rate and epsilon are parameters of your learning algorithm, and the discount factor is a property of the environment.

4 Theoretical questions (4 pt)

In this section we'll reflect on the main topics learnt in the RL unit in light of the practical exercises above. For each question please write a comprehensive yet parsimonious answer. Make sure to support your answer with either a rigorous argument or a counter example.

Question 8 (1 point)

What is the computational complexity of the VI algorithm, implemented in Question 1?

Question 9 (1 point)

In Question 3, the parameters govern the “riskiness” of the policy. Explain how each parameter affects the optimal policy.

Question 10 (2 point)

In Question 4, can you think about another exploration policy, instead of ϵ -greedy? How will this policy change the learning process (number of times an action is selected, variability of the estimated Q-values)?

Good Luck!