Intro to AI & AI, 2022 Spring
Reinforcement Learning

This program assignment is based on the project of UCB's AI course. You can refer to the full project via https://inst.eecs.berkeley.edu/~cs188/sp21/project6/. Here, we only try to implement "**Q-Learning, Epsilon Greedy, and Q-Learning and Pacman**. You have to use the source code from folder "rl_2022".

Problem Description:

- **Q-Learning:**

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.

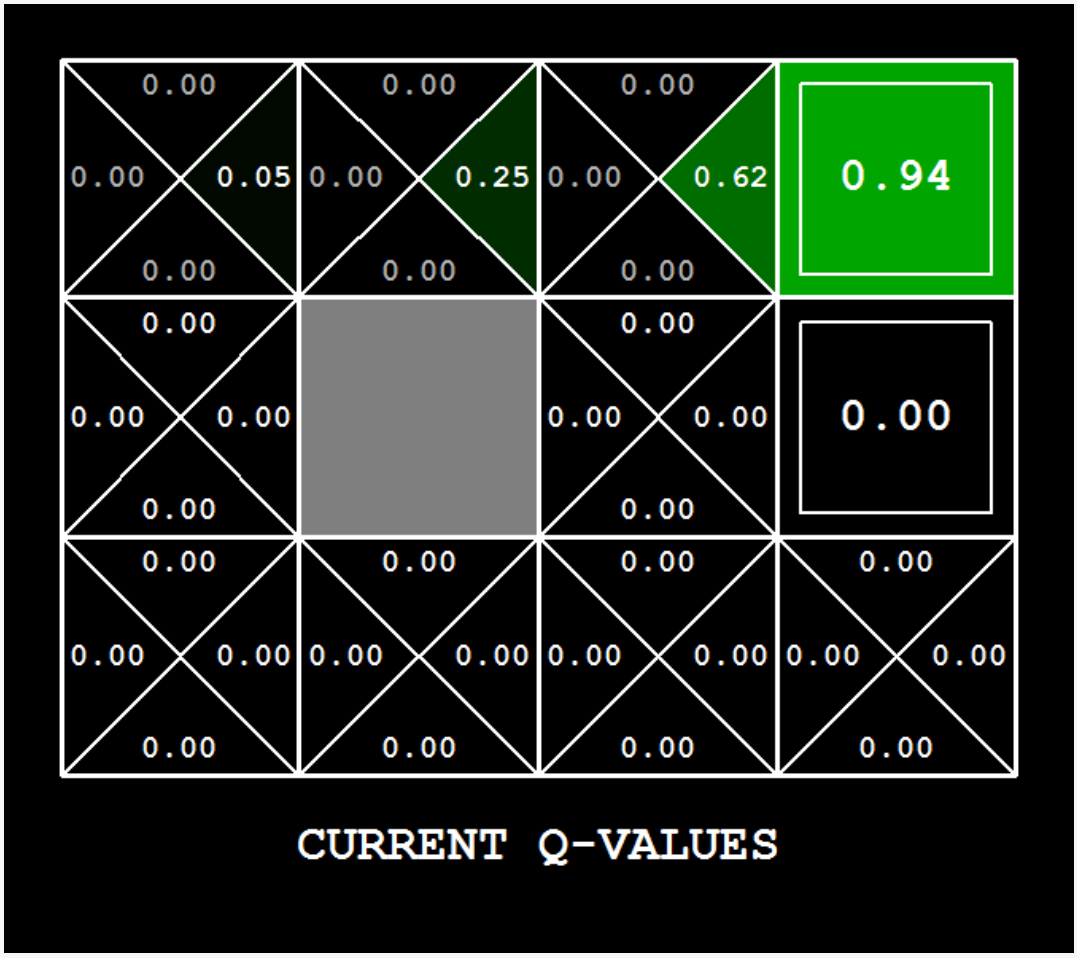
Important: Make sure that in your `computeValueFromQValues` and `computeActionFromQValues` functions, you only access Q values by calling `getQValue`. This abstraction will be useful for question 10 when you override `getQValue` to use features of state-action pairs rather than state-action pairs directly.

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

```
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 q1
```

- **Epsilon Greedy**

Complete your Q-learning agent by implementing epsilon-greedy action selection in `getAction`, meaning it chooses random actions an epsilon fraction of the time, and follows its current best Q-values otherwise. Note that choosing a random action may result in choosing the best action - that is, you should not choose a random sub-optimal action, but rather *any* random legal action.

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`.

After implementing the `getAction` method, observe the following behavior of the agent in gridworld (with epsilon = 0.3).

```
python 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 also observe the following simulations for different epsilon values. Does that behavior of the agent match what you expect?

```
python gridworld.py -a q -k 100 --noise 0.0 -e 0.1
```

```
python gridworld.py -a q -k 100 --noise 0.0 -e 0.9
```

To test your implementation, run the autograder:

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

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

```
python 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.

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.

- **Q-Learning and Pacman**

Time to play some Pacman! Pacman will play games in two phases. In the first phase, *training*, Pacman will begin to learn about the values of positions and actions. Because it takes a very long time to learn accurate Q-values even for tiny grids, Pacman's training games run in quiet mode by default, with no GUI (or console) display. Once Pacman's training is complete, he will enter *testing* mode. When testing, Pacman's `self.epsilon` and `self.alpha` will be set to 0.0, effectively stopping Q-learning and disabling exploration, in order to allow Pacman to exploit his learned policy. Test games are shown in the GUI by default. Without any code changes you should be able to run Q-learning Pacman for very tiny grids as follows:

```
python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l
smallGrid
```

Note that `PacmanQAgent` is already defined for you in terms of the `QLearningAgent` you've already written. `PacmanQAgent` is only different in that it has default learning parameters that are more effective for the Pacman problem (`epsilon=0.05, alpha=0.2, gamma=0.8`). You will receive full credit for this question if the command above works without exceptions and your agent wins at least 80% of the time. The autograder will run 100 test games after the 2000 training games.

Hint: If your `QLearningAgent` works for `gridworld.py` and `crawler.py` but does not seem to be learning a good policy for Pacman on `smallGrid`, it may be because your `getAction` and/or `computeActionFromQValues` methods do not in some cases properly consider unseen actions. In particular, because unseen actions have by definition a Q-value of zero, if all of the actions that *have* been

seen have negative Q-values, an unseen action may be optimal. Beware of the argmax function from util.Counter!

Note: To grade your answer, run:

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

Note: If you want to experiment with learning parameters, you can use the option `-a`, for example `-a epsilon=0.1,alpha=0.3,gamma=0.7`. These values will then be accessible as `self.epsilon, self.gamma` and `self.alpha` inside the agent.

Note: While a total of 2010 games will be played, the first 2000 games will not be displayed because of the option `-x 2000`, which designates the first 2000 games for training (no output). Thus, you will only see Pacman play the last 10 of these games. The number of training games is also passed to your agent as the option `numTraining`.

Note: If you want to watch 10 training games to see what's going on, use the command:

```
python pacman.py -p PacmanQAgent -n 10 -l smallGrid -a
numTraining=10
```

During training, you will see output every 100 games with statistics about how Pacman is faring. Epsilon is positive during training, so Pacman will play poorly even after having learned a good policy: this is because he occasionally makes a random exploratory move into a ghost. As a benchmark, it should take between 1000 and 1400 games before Pacman's rewards for a 100 episode segment becomes positive, reflecting that he's started winning more than losing. By the end of training, it should remain positive and be fairly high (between 100 and 350).

Make sure you understand what is happening here: the MDP state is the *exact* board configuration facing Pacman, with the now complex transitions describing an entire ply of change to that state. The intermediate game

configurations in which Pacman has moved but the ghosts have not replied are *not* MDP states, but are bundled in to the transitions.

Once Pacman is done training, he should win very reliably in test games (at least 90% of the time), since now he is exploiting his learned policy.

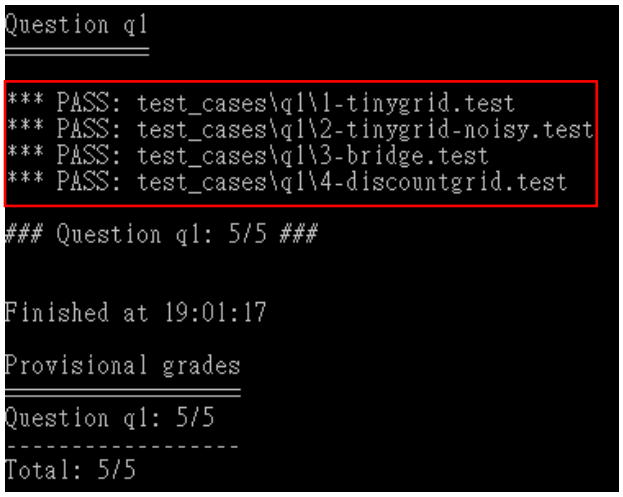
However, you will find that training the same agent on the seemingly simple `mediumGrid` does not work well. In our implementation, Pacman's average training rewards remain negative throughout training. At test time, he plays badly, probably losing all of his test games. Training will also take a long time, despite its ineffectiveness.

Pacman fails to win on larger layouts because each board configuration is a separate state with separate Q-values. He has no way to generalize that running into a ghost is bad for all positions. Obviously, this approach will not scale.

Requirements:

1. Your program should pass 3 tests by the autograder. (60%)

$ python autograder.py -q q1



$ python autograder.py -q q2

```
Question q2
===========

*** PASS: test_cases\q2\1-tinygrid.test
*** PASS: test_cases\q2\2-tinygrid-noisy.test
*** PASS: test_cases\q2\3-bridge.test
*** PASS: test_cases\q2\4-discountgrid.test

### Question q2: 2/2 ###


Finished at 19:02:39

Provisional grades
==================
Question q2: 2/2
----------------
Total: 2/2
```

$ python autograder.py -q q3

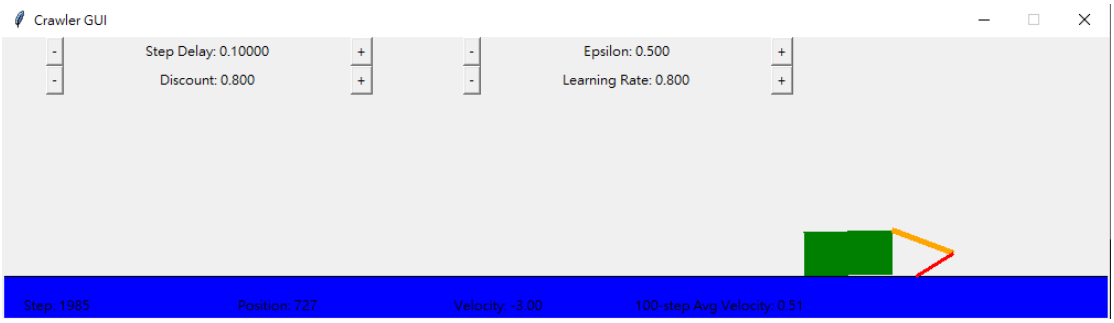
```
*** PASS: test_cases\q3\grade-agent.test (1 of 1 points)
***      Grading agent using command:  python pacman.py -p PacmanQAgent -x 2000 -n 2100 -l smallGrid -q -f --fixRandomSee
d
***      100 wins (1 of 1 points)
***         Grading scheme:
***          < 70:  0 points
***          >= 70:  1 points

### Question q3: 1/1 ###

Finished at 19:07:32

Provisional grades
==================
Question q3: 1/1
----------------
Total: 1/1
```

2. Your program should also work for the crawler program. (20%)



3. Write comments of your code to describe functionalities of your code segments and the reason why you implement it. (20%)
4. **DO NOT copy the code from others.**

Submissions:

1. qlearningAgents.py. (Your comments should be included.)

2. Any other files you modified.