A4: Reinforcement Learning

In this assignment, you'll implement q-learning for a grid world.

To make it more fun, we have a real GUI so you can see what your agent has learned. Unfortunately, that means we won't be using a notebook/Collab. So, you'll have to run this on your own machine.

The Gridworld class is already implemented for you in `gridworld.py`. You should be able to launch the gridworld interface with the command:

```
python gridworld.py
```

In a Gridworld, each state is a tuple of integers (x,y), corresponding to the coordinates on the grid. And for each non-terminal state, there are exactly four actions, going UP, DOWN, LEFT, or RIGHT. Gridworld also has two parameters, `noise` and `living_reward`.

`noise` defines the probability of the robot not doing exactly what you tell it to do. For example, if you tell the robot to go UP, the probability of it actually going up is 1 - noise; the probability of the agent going the perpendicular direction LEFT or RIGHT are both noise/2. Furthermore, if the robot is hitting a wall, then the outcome state will still be the same state, because the robot didn't move at all. By default, noise is 0.2.

`living_reward` defines the reward given to the robot for each action that leads to a non-terminal state. By default it's 0, so no reward given to the agent before reaching the terminal states.

All your code will go in `agents.py`. The only information you get from the game object is `game.get_actions(state: State)`, which returns a set of Actions possible from a given state.

You will also see that the GUI has an episode label. Each iteration is a single update, and each episode is a collections of iterations such that the agent starts from the starting state at the beginning of an episode and reaches a terminal state at the end of an episode.

Part 1: Q-learning

A stub of a Q-learner is specified in `QLearningAgent`. A `QLearningAgent` takes in

- game, an object to get a set of available actions for a state s.
- discount, the discount factor γ.
- learning_rate, the learning rate α.
- explore_prob, the probability of exploration ϵ for each iteration.

Note that the states in Gridworld aren't specified ahead of time. Your code should be able to deal with a new state and a new (state, action) pair.

Then you need to implement the following methods for this part:

- agent.get_q_value(state, action) returns the Q-value for (state, action) pair. Q(s,a) from Q-table. For a never seen pair, the Q-value should be 0.
- agent.get_value(state) returns the value of the state $V(s)$.
- agent.get_best_policy(state) returns the best policy of the state, $\pi(s)$.
- agent.update(state, action, next_state, reward) updates the Q-value for (state, action) pair, based on the value of the next_state and reward given.

Note: For `get_best_policy`, you should break ties randomly for better behavior. The `random.choice()` function will help.

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

- Run `python gridworld.py`
- To help with debugging, you can turn off noise by using the –noise 0.0 parameter in the command line (though this obviously makes Q-learning less interesting). If you manually steer up and then right along the optimal path for four episodes, you should see the Q-values in qlearning_sample.png .

Part 2: Epsilon Greedy

Complete your Q-learning agent by implementing epsilon-greedy action selection in `agent.get_action(state)`, meaning it chooses random actions an ϵ 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 Bernoulli trial with probability p of success by using `random.random() < p`, which returns `True` with probability p and `False` with probability $1 - p$.

Part 3: PacMan

For fun, I've also included a PacMan GUI. You should be able to train your agent in this new game without modifying your code. The command below will first train for 2000 games, then show the GUI when playing 10 games:

```
python pacman.py --train 2000 --play 10 small
```

Try it out and see if you can get your agent to play PacMan!

If your agent works for Gridworld but does not seem to be learning a good policy for Pacman on smallGrid, it may be because your `get_action` and/or `get_best_policy` 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!

[This assignment is adapted from one of Chris Callison-Burch's.]