This assignment asks you to implement Q-learning and play with it in a couple of domains. It is a slight adaptation of a project developed for the Berkeley AI class, available at http://inst.eecs.berkeley.edu/~cs188/fa09/projects/reinforcement/reinforcement.html.

The project is in python, and you will have to implement a set of methods in an existing python code base. If you have not used python before, Sam has written a python primer (primer.py), available on the Assignments page, that walks you through the basics of python. There is an excellent python tutorial here: http://docs.python.org/tutorial/.

You can install python on your computer from here: http://www.python.org/download/.

The Environment

This project uses the following python code base: http://inst.eecs.berkeley.edu/~cs188/sp09/projects/reinforcement/reinforcement.zip. Download this code and unzip it. The unzipped directory contains a number of python files.
You will only need to modify qlearningAgents.py. It will be useful to also look at (but not modify) mdp.py and learningAgents.py.

To get more familiar with the environment, the following command will run gridworld in manual control mode. You will see a gridworld domain. The blue dot is the agent. The gray squares are “walls,” and the black squares are ones to which the agent can move. Note that when you press up, the agent only actually moves north 80% of the time. This is noise that forces the agent to take a random action.

```
python gridworld.py -g MazeGrid -m
```

Without the -m parameter, you can watch the default agent moving around randomly.

```
python gridworld.py -g MazeGrid
```

To see all the possible options for gridworld, run the following command:

```
python gridworld.py -h
```
**Question 1**

(50 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`. For this question, you must implement the `update`, `getValue`, `getQValue`, and `getPolicy` methods.

**Hint 1:** For `getValue` and `getPolicy`, you should break ties randomly for better behavior. The `random.choice(list)` function chooses a random element from a list.

**Hint 2:** 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.

**Hint 3:** Make sure that you only access Q values by calling `getQValue` in your `getValue`, `getPolicy` functions, and not by reading the data structure values directly.

**Hint 4:** A very natural way to represent this in Python is to use a dictionary that maps states into lists of actions. See the Python primer for details on dictionaries and lists.

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

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

The `-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 2**

(15 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.

```
python gridworld.py -g MazeGrid -a q -k 100
```

**Hint:** 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 3**

(0 points, nothing to submit, try it just for fun) 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.
Question 4

(35 points) Time to play some Pac-Man! Pac-Man will play games in two phases. In the first phase, training, Pac-Man 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, Pac-Man’s training games run in quiet mode by default, with no GUI (or console) display. Once Pac-Man’s training is complete, he will enter testing mode. When testing, Pac-Man’s self.epsilon and self.alpha will be set to 0.0, effectively stopping q-learning and disabling exploration, in order to allow Pac-Man 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 Pac-Man 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 Pac-Man 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 last 10 runs.

Hint: If your QLearningAgent works for gridworld.py and crawler.py but does not seem to be learning a good policy for Pac-Man on smallGrid, it may be because your getAction and/or getPolicy 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.

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 Pac-Man is faring. Epsilon is positive during training, so Pac-Man 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 about 1,000 games before Pac-Man’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 Pac-Man, with the now complex transitions describing an entire ply of change to that state. The intermediate game configurations in which Pac-Man has moved but the ghosts have not replied are not MDP states, but are bundled in to the transitions.

Once Pac-Man 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’ll find that training the same agent on the seemingly simple mediumGrid may not work well. In our implementation, Pac-Man’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. Pac-Man 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.

Question 5

(Extra Credit 10 points) Implement an approximate q-learning agent that learns weights for features of states, where many states might share the same features. Write your implementation in ApproximateQAgent
class in qlearningAgents.py, which is a subclass of PacmanQAgent.

Note: Approximate q-learning assumes the existence of a feature function \( f(s, a) \) over state and action pairs, which yields a vector \( f_1(s, a) \ldots f_i(s, a) \ldots f_n(s, a) \) of feature values. Note: The \( f_i \)-s correspond to the \( \theta_i \)-s from our discussion in class. We provide feature functions for you in featureExtractors.py. Feature vectors are util.Counter (like a dictionary) objects containing the non-zero pairs of features and values; all omitted features have value zero.

The approximate q-function takes the following form:

\[
Q(s, a) = \sum f_i(s, a) \ast w_i
\]

where each weight \( w_i \) is associated with a particular feature \( f_i(s, a) \). In your code, you should implement the weight vector as a dictionary mapping features (which the feature extractors will return) to weight values. You will update your weight vectors similarly to how you updated q-values:

\[
w_i = w_i + \alpha \ast \text{correction} \ast f_i(s, a)
\]

\[
\text{correction} = R(s, a) + \gamma \ast V(s^{'}) - Q(s, a)
\]

Note that the correction term is the same as in normal Q-Learning.

By default, ApproximateQAgent uses the IdentityExtractor, which assigns a single feature to every (state,action) pair. With this feature extractor, your approximate q-learning agent should work identically to PacmanQAgent. You can test this with the following command:

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

Important: ApproximateQAgent is a subclass of QLearningAgent, and it therefore shares several methods like getAction. Make sure that your methods in QLearningAgent call getQValue instead of accessing q-values directly, so that when you override getQValue in your approximate agent, the new approximate q-values are used to compute actions.

Once you’re confident that your approximate learner works correctly with the identity features, run your approximate q-learning agent with our custom feature extractor, which can learn to win with ease:

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor
-x 50 -n 60 -l mediumGrid
```

Even much larger layouts should be no problem for your ApproximateQAgent. (warning: this may take a few minutes to train)

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor
-x 50 -n 60 -l mediumClassic
```

If you have no errors, your approximate q-learning agent should win almost every time with these simple features, even with only 50 training games. Congratulations! You have a learning Pac-Man agent!

**Submission Instructions**

Submit your qLearningAgents.py to the submit server by 11:59pm on Dec. 7.