CS 380: Artificial Intelligence

Assignment 4

Q-Learning (10 pts)

In this assignment, we will implement a basic Q-learning algorithm to learn a policy of behavior for a simple grid world. We will use a grid environment similar to that discussed in lecture, except slightly larger and with one dead-end path:

```
 -------
|       |
| ###  -|
| # #  +|
| # ####|
|       |
 -------
```

In this environment, the player is trying to reach the good location '+' while avoiding the bad location '–'. The '#' characters represent walls that cannot be crossed. The player also cannot move off the edge of this environment. We would like to learn a policy such that wherever a player is placed into this environment, the player can follow the policy to reach the good goal state.

As we know, Q-learning requires us to define a reward for states of the environment. We will assume a very simple reward scheme: +10 when the agent is at the goal location '+', –10 when it is at the bad location '–', and 0 everywhere else.

You are given the code to represent the environment, state, and actions. Your task will be to augment this code with the Q-learning algorithm that learns the Q table discussed in class—that is, the values for the table Q(S,A). The code provided is written in Python, and the code for this assignment should run on tux.cs.drexel.edu with a shell script, as we did for the previous assignments. Note that **you may only use built-in standard libraries (e.g., math, random, sys, etc.); you may NOT use any external libraries or packages** (if you have any questions at all about what is allowable, please email the instructor).

Implementation Setup

As before, this assignment requires a shell script that can pass an argument to your code—thus allowing you to use **python** or **python3** while allowing us to be able to run your code with a consistent interface. This is the same code as you used in the previous assignments, including the ability to accept two arguments in the general format:

```
sh run.sh <command> [<optional-argument>]
```

As before, this scheme will allow you to test your code thoroughly, and also allow us to test your code using a variety of arguments.

A sample shell script **run.sh** has been provided with this assignment. You should not need to modify this script file.

Environment, State, and Action Implementation

Also provided with this assignment is the Python file **qlearn.py**. The code provides an implementation for a number of components: an **Env** (environment) class that encodes a string into a grid representation; a **State** class that, given an environment, maintains the position of the agent within the environment; an **Action** class that represents an action as

a *dx,dy* movement of the agent; and some additional code that provides the default environment, actions, and main wrapper. As before, much of the code is left uncommented to allow you to practice reading and understanding such code.

Implementing the Q-Table

You should begin by filling in the code to represent the Q-table, that is, the table that will represent Q(S,A). As you know, the table should include a Q value for each state-action pair. In the case of our environment here, the state is itself an *x,y* pair (the location of the agent), so it might be most convenient to implement your table as a 3-dimensional array (*x*, *y*, action). The implementation of the related helper functions (**get_q()**, **get_q_row()**, **set_q()**) should perform these actions in whatever way is appropriate for your representation.

Learning in a Single Episode

The most critical part of this assignment is the implementation of the **learn_episode()** function. There are comments in the code template to help you structure this code. In summary, the learning episode should start at a random state in the environment, select a random legal action, and proceed to update the Q-table according to the equation we saw in lecture:

$$Q(S,A) \leftarrow (1-\alpha)Q(S,A) + \alpha[R + \gamma \max_{A'} Q(S',A')]$$

R is the reward after performing the action A in state S — in other words, the reward for the agent in state S'. The default values of α and γ are provided in the code and can be left as is. To compute the final *max* part of the equation, note that it is essentially the maximum of the row of the Q-table (which can be gotten with **get_q_row()**) for state S'.

Learning for Multiple Episodes

When your code successfully runs through a single episode, you can finally implement the **learn()** function that runs multiple episodes. As given in the provided code (in the main function), we will assume 100 training episodes for this assignment.

As one last piece, you should implement the **__str__()** function that converts your Q-table to a string, to allow for printing of the table values. The formatting of the table should match the formatting below; in particular, zero values should be printed as '----' (4 dashes); numeric values should be printed with 2 decimal places; and tab characters ('\t') should be used between table cells. The 4 tables corresponding to the 4 actions should be printed in the order provided in the **ACTIONS** variable defined at the top of the code.

When you are finished, you should be able to run the learning with the simple shell command "**sh run.sh learn**". For the run, please *print the board state for every move,* so that the user can see the successive states and see the 'A' agent character moving from space to space. In the end, your output should look like the following below:

```
> sh run.sh learn
 -------
|       |
| ###  -|
| # #  +|
| # ####|
|  A    |
 -------
 -------
|       |
| ###  -|
| # #  +|
| # ####|
| A     |
 -------
 -------
|       |
| ###  -|
| # #  +|
| # ####|
|A      |
 -------

<... many boards representing all your episode simulations ...>

UP
----  ----  ----  ----  ----  ----  ----
1.54  ----  ----  ----  2.97  3.29  ----
1.19  ----  ----  ----  3.85  3.84  ----
0.96  ----  0.33  ----  ----  ----  ----
0.77  ----  0.38  ----  ----  ----  ----

RIGHT
1.94  2.30  2.72  3.30  3.59  1.15  ----
----  ----  ----  ----  4.97  -9.85 ----
----  ----  ----  ----  7.11  9.02  ----
----  ----  ----  ----  ----  ----  ----
0.53  0.45  0.39  0.33  0.28  0.24  ----

DOWN
1.13  ----  ----  ----  4.13  4.83  -9.82
0.92  ----  ----  ----  5.29  6.40  ----
0.73  ----  0.39  ----  ----  ----  ----
0.60  ----  0.45  ----  ----  ----  ----
----  ----  ----  ----  ----  ----  ----

LEFT
----  1.53  1.83  2.14  2.18  2.85  2.14
----  ----  ----  ----  ----  3.31  ----
----  ----  ----  ----  ----  4.17  ----
----  ----  ----  ----  ----  ----  ----
----  0.64  0.53  0.45  0.39  0.33  0.28
```

This output gives the Q-table values for each of the 4 actions. For example, look especially at the cells around the good ('+') and bad ('–') locations. A DOWN action into the bad cell has a highly negative Q value (-9.82), whereas a RIGHT action into the good cell has a highly positive Q value (9.02). The values that lead to good and bad also reflect how good they are, with diminishing reward as you get farther from the goal.

As another example, look at the bottom row of each table: the largest values at each cell are for the LEFT action; the RIGHT action has lower but reasonable scores (since the agent can always go RIGHT and then LEFT); the DOWN action is always zero (since this is not a legal action); and the UP action is best on the leftmost side, but not very good for going up into the dead-end path.

Please note that **these values are generated by random simulations and thus your values will differ from those above**; in fact, they will differ on each run. But even when the exact numbers change, the same general pattern should emerge on every run — namely, that the highest action for any particular cell should be the action that ultimately leads toward the positive reward.

Academic Honesty

Please remember that you must write all the code for this (and all) assignments by yourself, on your own, without help from anyone except the course TA or instructor.

Submission

Remember that your code must run on **tux.cs.drexel.edu**—that's where we will run the code for testing and grading purposes. Code that doesn't compile or run there will receive a grade of zero.

For this assignment, you must submit:

- Your Python code for this assignment.
- Your **run.sh** shell script that can be run as noted in the examples.

Please use a compression utility to compress your files into a single ZIP file (NOT RAR, nor any other compression format). The final ZIP file must be submitted electronically using Blackboard—do not email your assignment to a TA or instructor! If you are having difficulty with your Blackboard account, you are responsible for resolving these problems with a TA or someone from IRT before the assignment it due. If you have any doubts, complete your work early so that someone can help you.