P3 - Mountain Car

Due Dec 6 by 2pm in the 366 dropbox in CSC.
Policy: This project can be done in teams of up to two students (all students will be responsible for completely understanding all parts of the team solution)

In this assignment you will implement Sarsa(λ) with tile coding to solve the mountain-car problem. Start with your existing code for running episodic Sarsa(0) that you wrote in p1 and for tile coding that you wrote in p2 (copy these files to a new directory and edit them into new versions). The code for the Mountain Car problem is available for download in MountainCar.java. The three actions (decelerate, coast, and accelerate) are represented by the integers 0, 1, and 2. The states are represented by arrays of two doubles corresponding to the position and velocity of the car.

MountainCar.java provides the same four functions as in the Party problem:

- `MountainCar.init()`, which takes no arguments and returns the initial state. In this case, the initial position is randomly chosen from [–0.6,–0.4] (near the bottom of the hill) and the initial velocity is zero.
- `MountainCar.numActions(s)`, which returns the number of actions available in state s. In this case, it always returns 3.
- `MountainCar.transition(s,a)` --> s', which returns a sample next state from state s given that action a is taken. Arrival in the terminal state is indicated by s' = `null` (as opposed to –1 for the Party problem).
- `MountainCar.reward(s,a,s')`, which returns a sample reward given a transition from s to s' under the action a. In this case, it always returns –1.

You will need to change your tile coder so that it covers the 2D state space for the car position and velocity as given in the textbook on page 215 (Section 8.4). To start with, use the following parameters:

- numTilings = 4
- shape/size of tilings = 9 x 9, scaled to that an 8x8 subset exactly fills the allowed state space
- $\lambda = 0.9$
- $\alpha = 0.05/\text{numTilings}$
- $\varepsilon = 0$
- initial weights = random numbers between 0 and –0.01

Note that $\gamma=1$ in this formulation of the problem and cannot be changed.

Your program should implement the following equations:

$$\delta_t = r_{t+1} + \gamma\theta_t^T\phi(s_{t+1},a_{t+1}) - \theta_t^T\phi(s_t,a_t)$$

$$\mathbf{e}_t = \gamma\lambda\mathbf{e}_{t-1} + \phi(s_t,a_t)$$

$$\theta_{t+1} = \theta_t + \alpha\delta_t\mathbf{e}_t$$

$$a_t = \begin{cases} \text{random action} & \text{with probability } \varepsilon \\ \arg\max_a \theta_t^T\phi(s_t,a) & \text{otherwise} \end{cases}$$

where θ, ϕ, and $\mathbf{e}$ are all vectors of the same length, n = the total number of features, which is, with the standard parameters above, 4 x (9 x 9) x 3 (numTilings x tilesPerTiling x numActions). Basically, you call your tile coder on the state to get a list of four tile indices. These are numbers between 0 and 4 x 9 x 9 – 1. If the action is 0, then these four are the places where ϕ is 1 (elsewhere 0). If the action is action 1, then you add 4 x 9 x 9 to these numbers to get the places where ϕ is 1. And if the action is 2 then you add twice that. Basically, you are shifting the 1 indices into a unique third of the feature vector depending on which action is specified. This will pick out a different third of the θ vector for learning about each action.

Once your code is working, try a run of 200 episodes. The initial episodes will be quite long, but eventually a good solution should be found wherein episodes are 120 steps long or less. After good performance is reached, make a 3D plot of minus the learned state values. That is, plot

$$-\max_a Q(s,a) \quad = \quad -\max_a \theta^T\phi(s,a)$$

as a function of the state, over the range of allowed positions and velocities as given in the book.

Now run 50 runs of 150 episodes each. Plot a learning curve like you did in p1 (average episode length vs. episode number).

What to turn in. Turn in your modified versions of Sarsa.java and Tilecoder.java, your 3D plot, and your learning curve.

Extra Credit

Experiment with changing the parameters from the values listed above to see if you can get faster learning or better final performance than is obtained with the original parameter settings. As an overall measure of performance on a run, use the total # steps in the first 150 episodes. If you can find a set of parameters that improves this performance measure by two standard errors, then you will earn an extra 5 points (out

of 72 total on the project). To show the improvement, you must do many runs with the standard parameters and then many runs with your parameters, and measure the mean performance and standard error in each case (a standard error is the standard deviation of the performance numbers divided by the square root of the number of runs minus one). If the difference between the means is greater than twice the larger of the two standard errors, then you have shown that your parameters are significantly better. It is permissible to use any number of runs greater or equal to 50 (note that larger numbers of runs will tend to make the standard errors smaller).

To collect your extra credit, report the alternate parameter settings that you found, the means and standard errors you obtained for the two sets of parameters, and the number of runs you used in each case.

Extra Extra Credit

Finally, once you have found your favorite set of parameters, make a learning curve based on 500 runs of 150 episodes with them. Provide a printout of this curve, along with is single average performance number and its standard error on these 500 runs. This will be used to compare your team's performance with that of the other teams. The top three teams will receive additional extra credit:

1st place: +9 points on the project
2nd place: +6 points
3rd place: +3 points