Homework 5
Reinforcement Learning

CMU 15-780: Graduate AI (Spring 2016)
OUT: March 15, 2016
DUE: 11:59pm March 23, 2016

Instructions

Collaboration/Academic Policy

You may discuss assignments with other students as you work through them, but writeups must be done alone. NO DOWNLOADING OR COPYING OF CODE OR OTHER ANSWERS IS ALLOWED. If you use a string of at least 5 words from some source, you must cite the source; however you cannot just copy the solution from some source but instead you should write it up in your own words.

Submission

Please create a tar archive of your answers and submit to Homework 5 on autolab. You should have two files in your archive: a completed `problems.py` for the programming portion, and a PDF for your answers to the written component. Your completed functions will be autograded by running through several test cases and their return values will be compared to the reference implementation. There is a `sample.txt` that contains sample inputs and outputs for reference. Please put your Andrew ID somewhere on the first page of your written answers.

You have 8 late days for homeworks over the semester, and can use at most 3 for one homework.

TAs

If you need help, the names beside the questions are the names of the TAs who came up with them, and are more likely to be familiar with the topics.

1 Written: Policy iteration convergence (Christian; 34 points)

Recall the *policy iteration* algorithm and notation from slide 22 on Markov Decision Processes. We will alter notation slightly here. Let the initial policy be $\pi_0 = \hat{\pi}$ and let π_i be the policy after performing update i.

1.1 17 pts

Prove the following equation:

$$v^{\pi_i} \leq v^{\pi_{i+1}}. \tag{1}$$

Hint: First show the following and use it in your proof:

$$r \geq (I - \gamma P^{\pi_{i+1}})v^{\pi_i}.$$

1.2 17 points

Using (1), prove that policy iteration converges to the optimal solution.

Hint: Use the Bellman optimality equation (see slide 13).

2 Programming: Q-Learning [Daniel; 66 points]

You will be implementing the Q-learning algorithm. First, you will implement some of the pieces and then combine them together to complete the Q-learning algorithm.

We will represent states as integers. If there are n states, then the states will be `[0,1,..., n-1]`. Actions will also be represented the same way i.e. if there are m actions, then the actions will be `[0,1,..., m-1]`. Q-Learning involves updating the Q-function $Q(s, a)$, which will be represented as a list of lists. For example `q = [[0,0],[0,0],[0,0]` can be an Q-function with 3 states and 2 actions initialized to zero; then `q[0][1]` would correspond to the Q-value for state 0 and action 1.

We will represent a policy as a list. For example `p = [0, 1, 0]` could correspond to a policy for an MDP with 3 states and 2 actions. In this case, `p[1]` would correspond to the action that is taken for state 1, which is 1.

We will be testing your algorithms with multiple runs on a few MDPs

2.1 Q-Learning Update [20 points]

Given a timestep (s, a, r, s') on which you were at state s, took action a, got reward r and transitioned to a new state s', you will compute an update to the given Q-function according to the Q-learning update formula. α and γ will also be given. You will complete the function `ql_update(q, alpha, gamma, s, a, r, ss)`.

2.2 Greedy Policy [20 points]

Given the current Q-function, you will return the greedy policy associated with it. If there is a tie between multiple actions for a particular state, use the action with the lowest index. You will complete the function `ql_policy(q)`.

2.3 Q-Learning Iteration [20 points]

Here, you will combine your updates and the policy in order to run one iteration of the Q-learning algorithm. The iteration will be given a new data point consisting of (s, a, r, s'). Then you should use your Q-learning update to update the Q-function. Then you should use the epsilon-greedy policy, built on the greedy policy from the updated Q-function, in order to return the next action to take.

You should use 0.3 for α in the Q-learning update and 0.1 for ϵ in the epsilon-greedy policy.

You will complete the function `ql_iteration(q, gamma, s, a, r, ss)`.

2.4 Q-learning Tuning [6 points]

Here, you should play around with α and ϵ to try to get a better average total reward (we will use multiple runs and average the total reward to account for stochasticity). ϵ doesn't need to be a constant and can depend on the given iteration number. You will pass a test case as long as this algorithm outperforms the baseline by a statistically significant amount; the baseline is the algorithm in the previous part.

You will complete the function `ql_iteration_tuned(i, q, gamma, s, a, r, ss)`.