Reinforcement Learning
Winter 2017
Assignment 2 - Q-learning

Note: this assignment is a slightly extended version of the assignment proposed here:

http://ai.berkeley.edu/reinforcement.html#Welcome

where we propose additional tasks, but also a significantly larger amount of coding hints.

Please have an environment-friendly policy, avoid printing these sheets !

1. **Q-Learning**

   Note that the value iteration agent deployed in the previous assignment does not actually learn from experience. Rather, it ponders its MDP model to arrive at a complete policy before ever interacting with a real environment. When it does interact with the environment, it simply follows the precomputed policy (e.g. it becomes a reflex agent). This distinction may be subtle in a simulated environment like a Gridword, but it's very important in the real world, where the real MDP is not available.

   We will now turn to Q-learning, i.e. we will stop assuming that a model of the game is available to us to build the value or Q-value functions. You will write a Q-learning agent, which does very little on construction, but instead learns by trial and error from interactions with the environment through the method **update**(state, action, nextState, reward).

   The core algorithm is the one detailed on slide 38 of L5. Here the policy applied in the game will be computed from:

$$a = \max_{a'} Q(s,a)$$

   and the learning algorithm will be:

For given state s, action a, next state s_+, reward r, do

1: $\delta \leftarrow r + \gamma \max_{a'} Q(s_+, a') - Q(s,a)$
2: $Q(s,a) \leftarrow Q(s,a) + \alpha\delta$

   You will develop your Q-learner in the class **QLearningAgent** that you can find in the file **qlearningAgents.py**. You will need to edit the following methods:

   1. The class initialization **__init__** where you simply need to decide on structures for storing your learned Q-value function. You will want to attribute your Q-value function to the object of the class (i.e. define e.g. a self.qvalue structure adequate for our purposes). You need to think here that we do not even know beforehand the states to which our agent will be exposed, we will figure that out on-the-fly !!
   2. The method **update** is where the Q-learning happens, and the core of our agent. It receives a state, an action, the next state and the reward obtained in this transition. From these information it must update the Q-value function, following a Q-learning algorithm. Here are a few useful remarks:
      - Here you want to think carefully on how you want to organise the data in your Q-value function. Think that for a given state you will need to interrogate your Q-value function to decide on the best actions.
      - As said before, you will need to handle the case of a state that has not yet been discovered by the agent and which is therefore not "present" yet in the Q-value function.

- Even though states are not know a priori by our agent, once a state is visited the agent can be interrogated for admissible action at a given state, i.e. the method self.getLegalActions(state) can always be invoked.

The method does not return anything, your Q-value function ought to be attributed to the self with a name of your choice, to be used in the other methods of the class.

3. Method **getQValue** must interrogate your Q-value function to return the Q-value of a given state and action. For an unknown state the method should return a value of 0.
4. Method **computeActionFromQValues** computes for a given state the best action according to the current Q-value function. It is important here to treat the "tied" case properly, i.e. the case of several actions having the same (best) value (which can happen when e.g. not all the actions from a given state have been tried). In the tied case, you ought to select a random action in the list of actions having the (same) highest value. It will be useful to use the method **choice** in the imported class **random** (visible throughout your q-agent class), i.e. the command random.choice(*some list*) return one of the items in the list at random. Breaking tied cases is a bit tricky to code elegantly. You may want to think well how to do it right. **Make sure you only access your Q-value function by calling getQValue !!** This is important for your code to run on the more advanced games.
5. Method **getAction** computes the action to take for a given state. The best action can be obtained by interrogating your own function computeActionFromQValues. We will modify this basic function in the next question.
6. Method **computeValueFromQValues** computes the value function of a given state, i.e. it computes $\max_a Q(s,a)$. If there are no legal action for that state, then it should return 0. **Make sure you only access your Q-value function by calling getQValue !!** This is important for your code to run on the more advanced games.

You can play with your Q-learner learn using manual control by prompting:

python gridworld.py -a q -k 5 -m

and using the arrows on your keyboard to navigate the game.

The option -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".

Hint: to help with debugging, you can turn off noise by using the –n 0.0 parameter (though this obviously makes Q-learning less interesting). If you manually steer the token north and then east along the optimal path for **four episodes and without noise**, you should see the following Q-values:

You can assess your code by running

python autograder.py -q q4

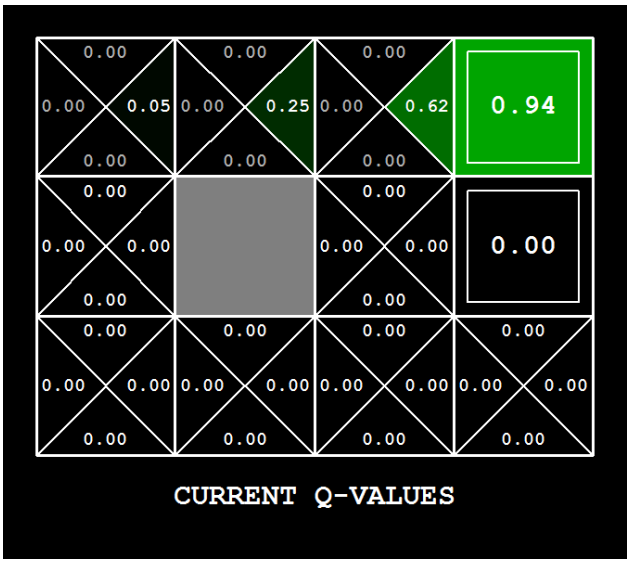

Figure 1:

2. **ε-greedy policy**

   We will now modify the method **getAction** so that it does not necessarily return the best action, but also takes random ones with a given probability ε (given by the attribute self.epsilon). The update algorithm will remain the same as before though!

   Modify your method **getAction** so that it picks a random action in the list of admissible actions (for the given state) with probability self.epsilon. You can use the method util.flipCoin(*probability of true*) to decide whether to take the best action or a random one. If there are no legal action for a give state, then your function must return None (existing type)

   You can test your implementation by running

   python gridworld.py -a q -k 100

   Your final Q-values should resemble those of your value iteration agent, especially along well-traveled paths. However, your average returns will be lower than what the Q-value function predict because of the random actions and the initial learning phase.

   To assess your code, run the autograder:

   python autograder.py -q q5

   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.

   This will invoke the crawling robot from class using your Q-learner. Play around with the various learning

parameters to see how they affect the agent's policies and actions. Note that the step delay is a parameter of the simulation, whereas the learning rate and epsilon are parameters of your learning algorithm, and the discount factor is a property of the environment.

3. **Bridge Crossing Revisited**

   We will not try your Q-learner on the bridge-of-death!!

   First, train a completely random Q-learner with the default learning rate on the noiseless BridgeGrid for 50 episodes and observe whether it finds the optimal policy. You can do that by running:

   python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1

   Now try the same experiment with $\varepsilon = 0$, i.e.

   python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 0

   Is there an epsilon and a learning rate for which it is highly likely (greater than 99%) that the optimal policy will be learned after 50 iterations? Provide the answer in **question6()** in the class **analysis.py**. It should EITHER return the values that make this possible OR the string 'NOT POSSIBLE' if there is none. Epsilon is controlled by -e, learning rate by -l.

   Note: Your response should be not depend on the exact tie-breaking mechanism used to choose actions. This means your answer should be correct even if for instance we rotated the entire bridge grid world 90 degrees.

   To grade your answer, run the autograder:

   python autograder.py -q q6

Questions

- Discuss briefly your experiments with the crawler.
- Detail your reasoning in the answers to question 3.
- Comment on the use of the ε-greedy policy vs. the best-policy approach? What does it do to the training process?
- Imagine and test in your code a very simple way to promote the exploration of unvisited states. What did you do and why?
- What does your experience with the Bridge-crossing problem tell you about the exploration problem in Reinforcement Learning? At what level should this problem be addressed? Can you imagine how?
- In this assignment, we have implemented Q-learning. In what sense is it different than deploying SARSA(0) in both question 1 and 2?