Reinforcement Learning

Eric Gu
December 4, 2018

For full code and notes, see “(CS 4641) Assignment 4 – Reinforcement Learning.ipynb”.

Written in Jupyter Notebook using OpenAI's Gym environments, and with algorithm implementation borrowed from Moustafa Alzantot and Arthur Juliani.

Description of Problems

Frozen Lake

Taken from OpenAI's description of FrozenLake-v0: "The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile."

The surface is described using a grid like the following:

```
    SFFF       (S: starting point, safe)
    FHFH       (F: frozen surface, safe)
    FFFH       (H: hole, fall to your doom)
    HFFG       (G: goal, where the frisbee is located)
```

An episode ends when the agent either reaches the goal or falls in a hole. You receive a reward of 1 if you reach the goal, and zero otherwise.

Immediately, we should note that although the state space of this stochastic MDP is small (4x4=16), at each step, there is only a 1/3 chance going where you want to go, and a 1/3 chance you go to the left or right instead. This makes the map treacherous, since to successfully navigate to the goal, the agent has to thread a narrow path between 2 holes no matter what. On the upside, the problem is very simple because it has only one goal state.

Taxi

There are four designated locations in the grid world indicated by R(ed), B(lue), G(reen), and Y(ellow). When the episode starts, the taxi starts off at a random square and the passenger is at a random location. The taxi drives to the passenger's location, picks up the passenger, drives to the passenger's destination (another one of the four specified locations), and then drops off the passenger. Once the passenger is dropped off, the episode ends.

The grid world looks like the following:

```
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

Actions

There are 6 discrete deterministic actions:

- 0: move south
- 1: move north
- 2: move east
- 3: move west
- 4: pickup passenger
- 5: dropoff passenger

Rewards

There is a reward of -1 for each action and an additional reward of +20 for delivering the passenger. There is a reward of -10 for executing actions "pickup" and "dropoff" illegally.

Although the grid world appears to be 5x5, there are actually 500 discrete states because the passenger can be in 5 possible locations (including in the taxi) and there are 4 possible destination locations. The Taxi Problem is an example of a hierarchical MDP—a more complex problem involving subgoals (first find path to passenger, then path from passenger to destination) and permutations of destinations & passengers can be abstracted by encoding relevant subproblem information into the state. While this MDP is deterministic, it poses a challenge because of the far larger state space.

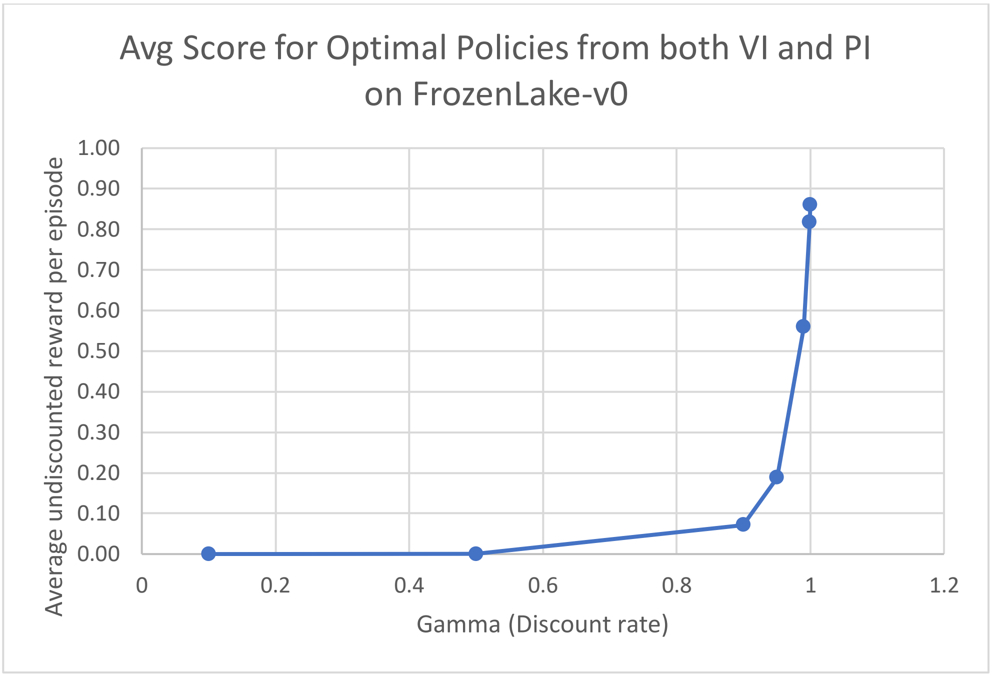
Let’s see how our algorithms perform.

Frozen Lake

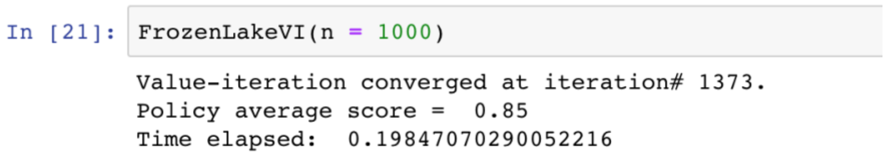
From what we know, we expect PI to converge in fewer iterations than VI, and in probably less wall clock time because of the greater computational requirements required by VI. We will see later if this holds true for both problems.

Value Iteration (VI)

We initialize the value functions to arbitrary random values. Let's start by running VI with a variety of values of gamma (discount rate):



We can graph the average undiscounted score for 100 episodes for each value of gamma—that is, the y-axis represents the proportion of episodes that reach the goal successfully. We find that VI performs best at gamma = 1.0, which makes sense—there is only one goal state, with no other rewards or punishments that are closer or farther away, so we want to incentivize the algorithm to always play the “long game.” Using gamma = 1.0, we can run value iteration, extract the solution policy from the optimal values, and evaluate that policy by running it on the environment 1000 times.



```
In [21]: FrozenLakeVI(n = 1000)

         Value-iteration converged at iteration# 1373.
         Policy average score =  0.85
         Time elapsed:  0.19847070290052216
```

Value iteration for Frozen Lake always converges at step #1373. The policy extracted from the optimal values has an average score of 0.85 over 1000 episodes and takes 0.198 seconds to run.

Policy Iteration (PI)

The performance of PI against different values of gamma will be the same as VI's, so we use the same hyperparameters as in VI and compare the resulting scores.

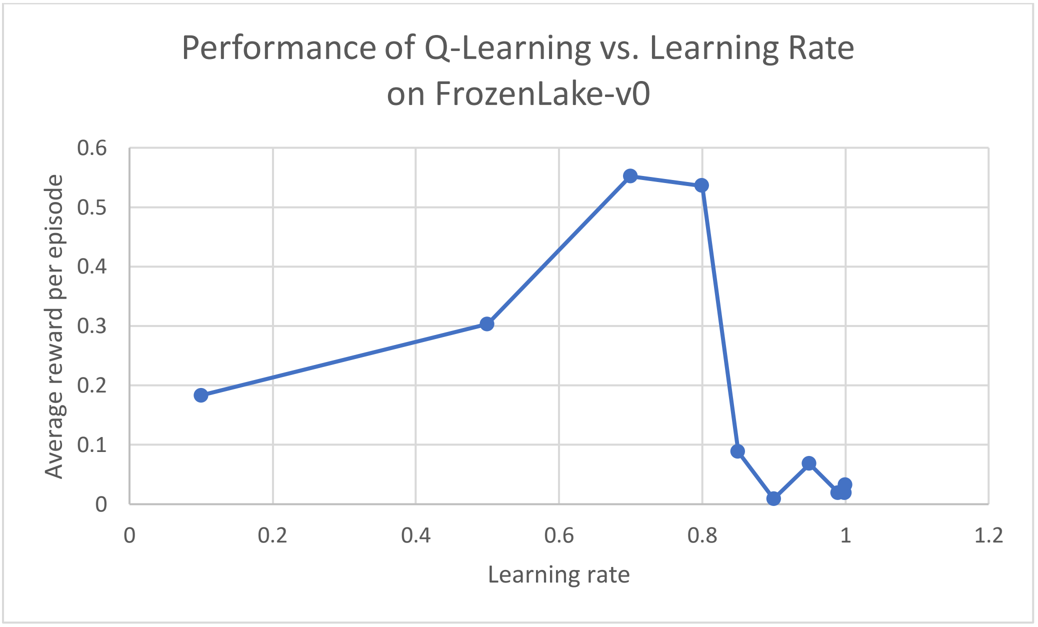
```
In [25]: FrozenLakePI(n = 1000)

         Policy-Iteration converged at step 4.
         Average scores =  0.85
         Time elapsed:  0.09478203854814637
```

Not too surprising! PI converges in only 4 steps. When evaluating the optimal policy provided by the algorithm, PI has the same average score of 0.85 across 1000 episodes, but only requires 0.0948 seconds to run. The policies are the same for both algorithms. Here, the principle advantage over VI is the speed at which the algorithm converges on a solution. PI seems to solve Frozen Lake about twice as quickly as VI.

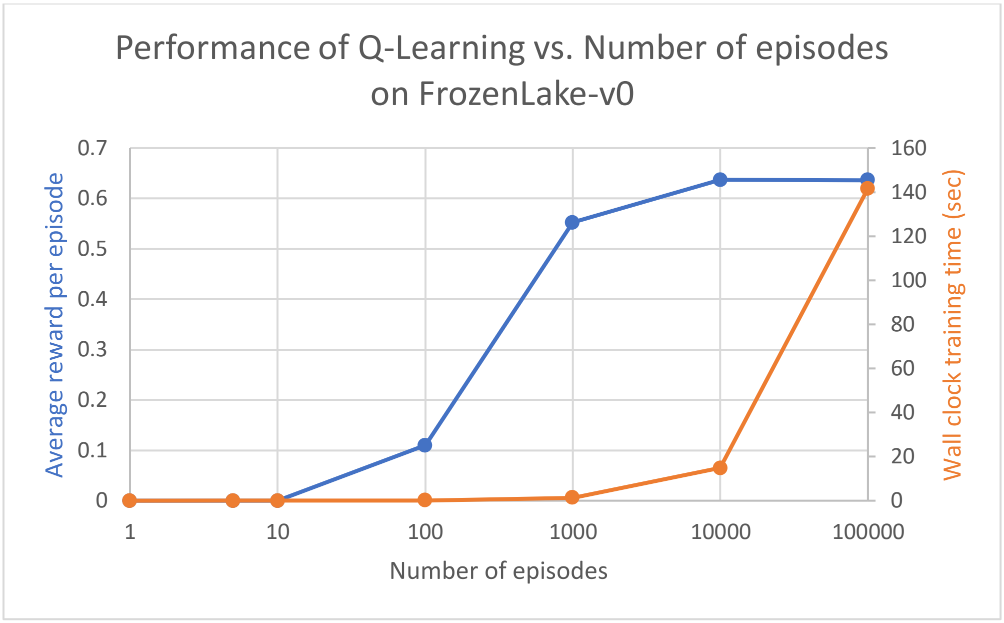
Q-Learning

Given the small state space, we can use a simple table implementation of Q-Learning to store the values for how good it is to take a given action within a given state. In the case of the Frozen Lake environment, we have 4 possible actions (the four directions of movement) for each of 16 possible states, giving us a 16x4 table of Q-values. We start by initializing the table to be uniform (all zeros), and then as we observe the rewards we obtain for various actions, we update the table accordingly.

Instead of using an epsilon-greedy or "epsilon-decreasing" strategy, we adopt a slightly modified, more robust exploration strategy that adds noise to the greedy selection of the next action from the Q-table in proportion to the size of the Q-values, and which decreases in later iterations as time goes on (source). For the sake of simplicity, we keep the learning rate constant in each episode.



After running the Q-Learning algorithm for 1000 iterations for the following learning rates: [0.1, 0.5, 0.7, 0.8, 0.85, 0.9, 0.95, 0.99, 0.999, 1.0], we find that the algorithm performs best with a (fixed) learning rate = 0.70 when gamma = 1.0. Next, we graph a learning curve for Q-Learning.



Since Q-Learning is an online algorithm, it doesn't stop on a definitively optimal solution because it is blind to the MDP's true reward and transition functions. Instead, by running 100,000 iterations, the Q-Learning algorithm has accumulated an average score of 0.63671 by learning approximations of the "correct policy" from the history of the rewards it receives by making certain actions in each state. However, these episodes for Q-Learning generally take more steps to reach the goal state than those from VI or PI because the environment does not punish longer routes, but only rewards reaching the goal state. It is apparent that the training time increases proportionately to the number of iterations, but the asymptotic performance shows a learning curve that has plateaued. Running the algorithm for only 10,000 iterations results in an average score of 0.637, virtually identical to that of 100,000 iterations, but takes 14 seconds compared to the 141 seconds for 100,000 iterations.

Just as a sanity check, we can try Q-Learning using epsilon-greedy fixed at 0.10, without decreasing epsilon over time. The performance *should* be worse than the above Q-Learning algorithm, where we began with a large epsilon (1.0) and asymptotically decreased it towards zero in order to prioritize exploration early on and exploit what the algorithm learned in later episodes.

```
In [11]: FrozenLakeQLearn_EpsilonGreedy()

         Score over time: 0.1215
         Time elapsed:  4.52724552503787
```

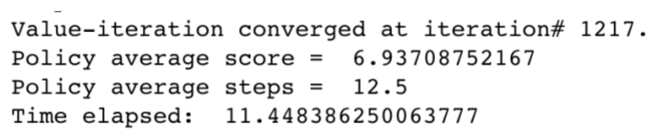
And indeed, we find that this overly naive approach to Q-Learning results in a lackluster performance of 0.1215 over 10,000 iterations.

Taxi

As mentioned, the Taxi Problem includes a larger state space and a longer time horizon between start and goal states, with steady punishment for taking longer, inefficient routes to passengers and destinations. We want to incentivize our algorithms to pursue the "long game"—that is, to see past the upfront penalties for taking lots of steps to eventually reach the +20 points for a successful dropoff—so we want a discount rate (gamma) close to 1.0. If we set gamma close to 0, the algorithm will instead be incentivized to make the agent wander around aimlessly before eventually entering an absorbing state for -10 points, which would be heavily discounted so as not to matter much at all. That said, we want to use gamma < 1.0 for practical reasons, as gamma = 1.0 can possibly result in an infinite horizon for this problem.

Value Iteration

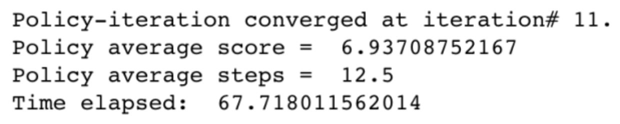
For the sake of evaluating the average scores of the solution policy, we will increase the margin of error for convergence from 1e-20 to 1e-4 so that the algorithm will converge within a reasonable amount of time.



```
Value-iteration converged at iteration# 1217.
Policy average score =  6.93708752167
Policy average steps =  12.5
Time elapsed:  11.448386250063777
```

When we run value iteration on gamma = 0.99, it converges in 1217 iterations and 11.45 seconds wall clock time. Evaluating the resultant policy 100 times gives an average 12.5 steps and score of 6.937. Without directly comparing to the results for PI, we can imagine that the solution policy generally takes the taxi through the optimal routes for each of the 20 subproblems, which take average 12.5 steps to complete, or about -13 points. That gives us an average score around 7 from successfully delivering the passenger to the destination.

Policy Iteration

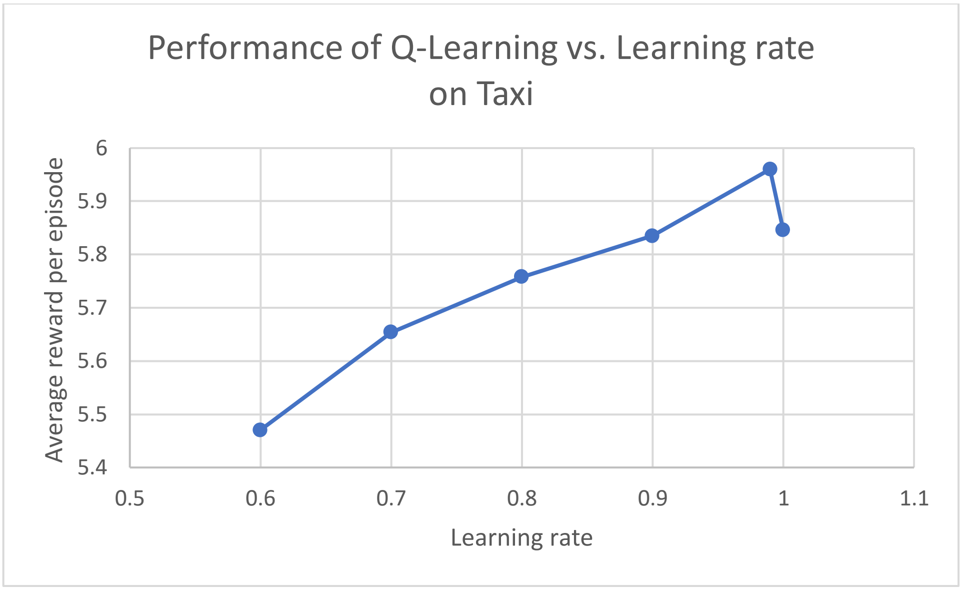


```
Policy-iteration converged at iteration# 11.
Policy average score =  6.93708752167
Policy average steps =  12.5
Time elapsed:  67.718011562014
```

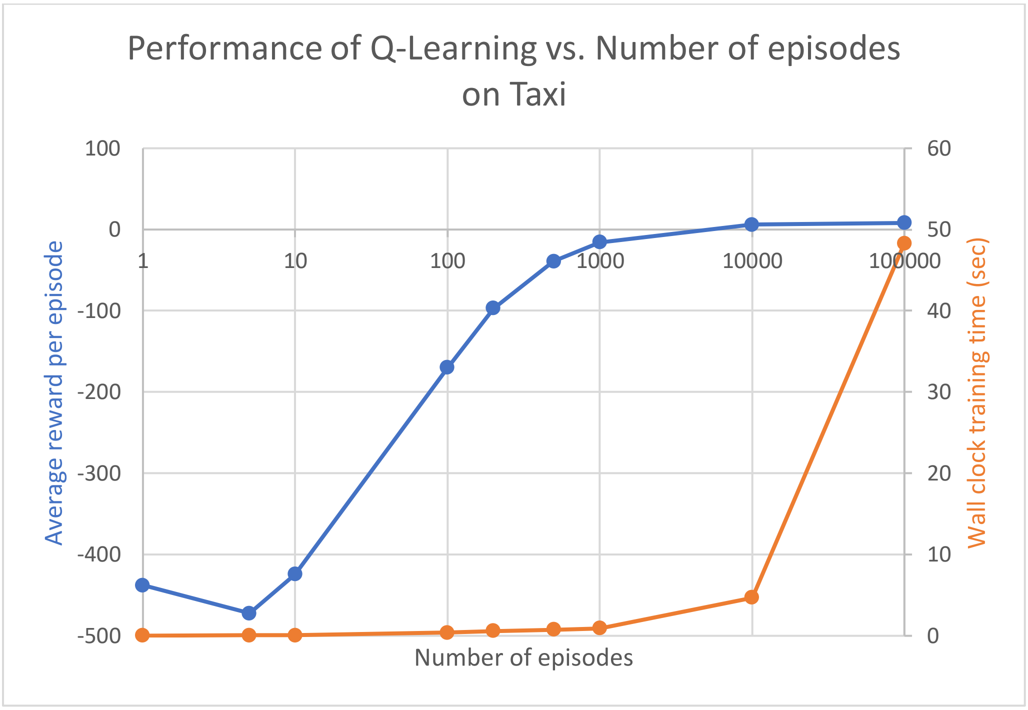
Of course, the solution policy from PI is the same, and so it has the same average steps and average score as VI. However, policy iteration on gamma = 0.99 converges in 11 iterations, but 67.72 seconds wall clock time—over 5 times as long as VI! This is because policy evaluation implies solving the Bellman equation typically through a system of equations, which is very costly for large state-action sets. Although PI converges faster than VI in many scenarios including Frozen Lake, in this case, the additional computational complexity of the inner loop made PI converge slower than VI in real time.

Q-Learning

For the sake of comparison, we will use the same Q-Learning algorithm implementation as we did for Frozen Lake, including the fixed learning rate and exploration strategy of greedily choosing from the Q-table with noise that decays as iterations increase. This algorithm will have a Q-table of 500 (size of state space) × 6 (size of action space) = 3000. We choose to train for 10,000 iterations in order to hypothetically cover every state-action pair plenty of times when graphing performance against learning rate. Because our choice of gamma does not negatively affect time to convergence in Q-Learning, we will choose to use gamma = 1.0.

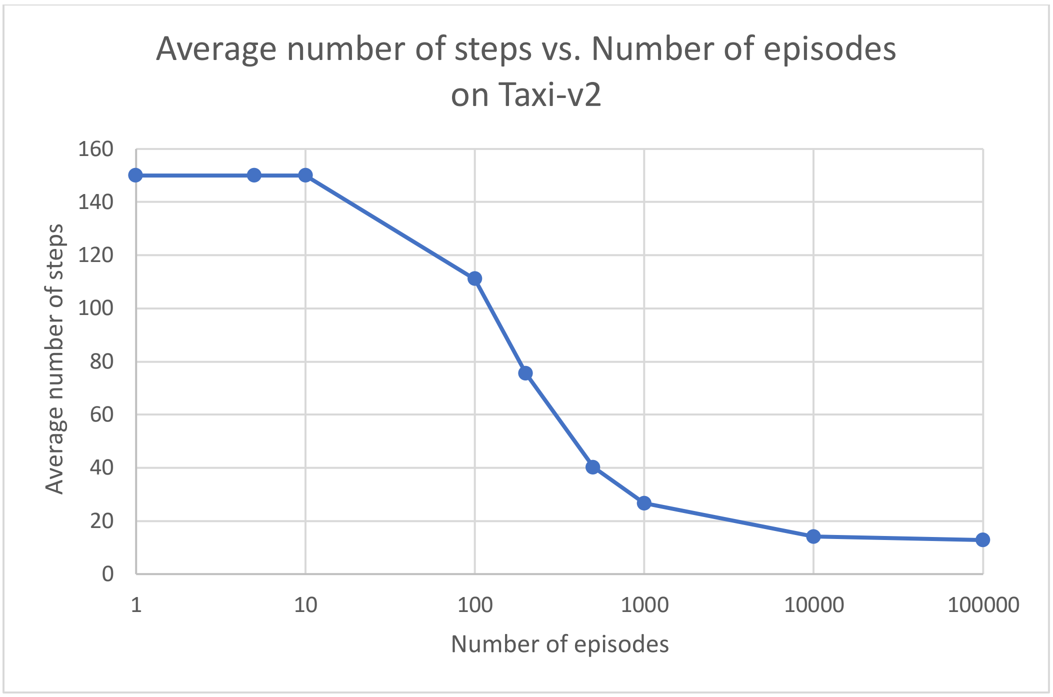


The average steps for each learning rate are all around 14, but the average reward per episode peaks sharply at learning rate = 0.99. Graphing the performance below, we see that the average scores of 10,000 or 100,000 cumulative episodes are between 5.9 and 8.1, plateauing thereafter. This is expected behavior, and closely mirrors what was seen in Q-Learning for Frozen Lake, since the RL algorithm can only asymptotically approach the optimal policy based on the values in the Q-table from episodes it has trained on.



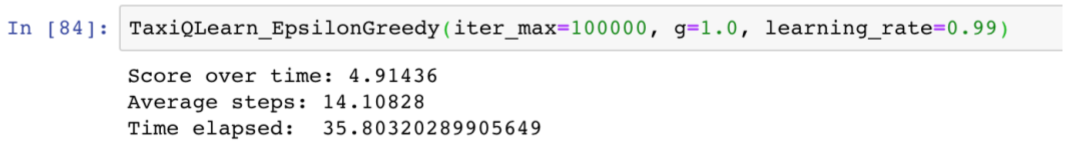
However, it is puzzling that Q-Learning is able to average a score of 8.1 over 100,000 episodes when VI and PI converged on an optimal policy that only evaluates to 6.9. We can attribute this discrepancy to the use of gamma = 1.0 on Q-Learning but gamma = 0.99 on VI and PI.

This phenomenon of Q-Learning approaching and possibly reaching the optimal policy found by VI and PI can also be seen in the average number of steps for each iteration:



When Q-Learning runs for 100,000 episodes, the cumulative number of steps averages to 12.81 per episode, quite close to the 12.5 steps in VI and PI. Relative to the Q-Learning algorithm trained for Frozen Lake, the Taxi algorithm appears to have done a better job of approximating the optimal policy found by VI and PI.

Finally, if we contrast with a simple epsilon-greedy approach to selecting actions in the Q-Learning algorithm where epsilon stays fixed at 0.05, we see that the performance is worse for the same reasons as in Frozen Lake—the algorithm is unable to take advantage of its training time to more fully explore the state-action space early on, nor is it able to fully exploit the information it has gained later on. It results in an average score of only 4.914, contrasted with 8.092 seen in the above greedy noisy Q-Learning algorithm.


```
In [84]: TaxiQLearn_EpsilonGreedy(iter_max=100000, g=1.0, learning_rate=0.99)

         Score over time: 4.91436
         Average steps: 14.10828
         Time elapsed:  35.80320289905649
```