Group U Frozen Lake

Cong Peng 210064410, Qinlin Xie 210171822, Qinjian Shangguan 210273126

School of Electronic Engineering and Computer Science, Queen Mary University of London, UK

c.peng@hss20.qmul.ac.uk, q.xie@hss20.qmul.ac.uk,

q.shangguan@hss20.qmul.ac.uk

1. Explain how your code for this assignment is organized. Did you make implementation decisions that deviate significantly from what we suggested?

To implement function *p* in the frozen lake environment, which returns probability of transitioning from state to next state given action, a three-dimensional array is used to store all the values and initialized in the *init* function of the frozen lake class. First of all, all the special situations are assigned with value *1.0* when current state is absorbing, in a hole or at destination, the next state must be absorbing state. Secondly, for regular situations, a coordinate composed of an index of row and another index of column is used to represent the position of the state of the lake. And then the new position will be calculated with the different slip action, the state will change only when the new position is still in the grid. For each slip action, the probability will add *self.slip / n\_actions*. And when the slip action is the same to the original one, the probability will add another value which is the probability of no slip. With the application of coordinates and traversal of action and slip action iterations, the code is readable and implemented elegantly.

For function *r* which returns the expected reward in having transitioned from state to next state with given action, no array is used before the function is called because the logic of reward is simple. The reward is *1* only when the agent takes any action at destination state. Otherwise, the reward is always *0*. In the code implement, *self.lake\_flat[state] == "$"* is used to represent this special situation. However, array *self.lake\_flat* does not include the absorbing state. Therefore, it is excluded before with condition *state == self.absorbing\_state*.

1. How many iterations did policy iteration require to find an optimal policy for the big frozen lake? How many iterations did value iteration require? Which algorithm was faster?

They both need *19* iterations to find an optimal policy for the big frozen lake. However, value iteration performs better than policy iteration from iteration *7* to *17* during the test. When the iteration is small, there exists no difference between the two algorithms’ performance. With the iteration increasing, value iteration achieves higher values of policy but both of them converge to an optimal one when finally reaching a specific iteration. To conclude, value iteration algorithm is still faster than policy iteration.

1. How many episodes did Sarsa control require to find an optimal policy for the small frozen lake? How many episodes did Q-learning control require? **Hint:** you may use policy evaluation to compare the value of each policy obtained by these algorithms to the value of an optimal policy.

From 2000 to 125, I have tried binary search to get the closet value.

When max\_episodes = 125, no algorithm can generate an available answer.

However, when the max\_episodes = 219, sarsa can find a solution, q\_learning still cannot.

When using binary search to 247, q\_learning can find a valid solution too.

1. In linear action-value function approximation, how can each element of the parameter vector ***θ*** be interpreted when each possible pair of state s and action a is represented by a different feature vector ***φ***(s, a) where all elements except one are zero? Explain why the tabular model-free reinforcement learning algorithms that you implemented are a special case of the non-tabular model-free reinforcement learning algorithms that you implemented.
2. Try to find an optimal policy for the big frozen lake by tweaking the parameters for Sarsa control and Q-learning control (maximum number of episodes, learning rate, and exploration factor). You must use policy evaluation to confirm that the resulting policy is optimal. Even if you fail, describe your experience.

To discover a pattern on how to tweak the maximum number of episodes(max\_episodes), learning rate(eta) and exploration factor (epsilon). I have tried each parameter individually and subtracted the value with the value get by policy\_evaluation. And then get the mean of the abstract delta matrix.

delta = (value - policy\_evaluation(env, optimal\_policy, gamma, theta, max\_episodes))

np.absolute(delta).mean()

repeat this step several times, and then get 3 graphs shown below,

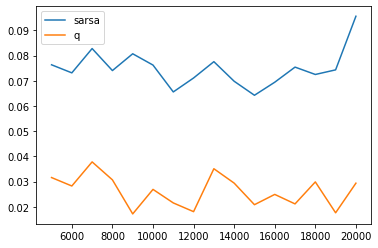
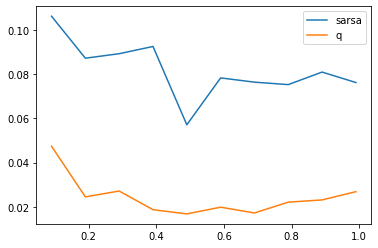


Fig.1 mean\_delta changed with max\_episodes (max\_episodes changed from 5000 to 20000 with step 1000) eta = 0.99 epsilon = 0.99

From this figure, we can tell that max\_episodes has little influence on the optimal policy got by these 2 algorithms. And the q\_learning method is generally better than sarsa.

  
Fig.2 mean\_delta changed with eta (eta changed from 0.09 to 0.99 with step 0.1) max\_episodes = 10000 epsilon = 0.99

From this figure, we can tell that with the increase of learning rate, both sarsa and q\_learning get better performance (get a relatively low mean\_delta). And, q\_learning is generally better than sarasa.

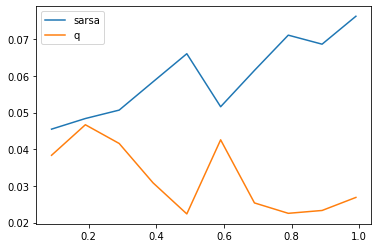


Fig.3 mean\_delta changed with epsilon (epsilon changed from 0.09 to 0.99 with step 0.1) max\_episodes = 10000 eta = 0.99

This figure is very interesting to us because, with the exploration rate increase, sarsa has a much worse performance than the q\_learning. I think this is kind of because sarsa is relatively conservative and q\_learning is more aggressive. So sarsa has a relatively better performance in a low exploration rate environment.

References