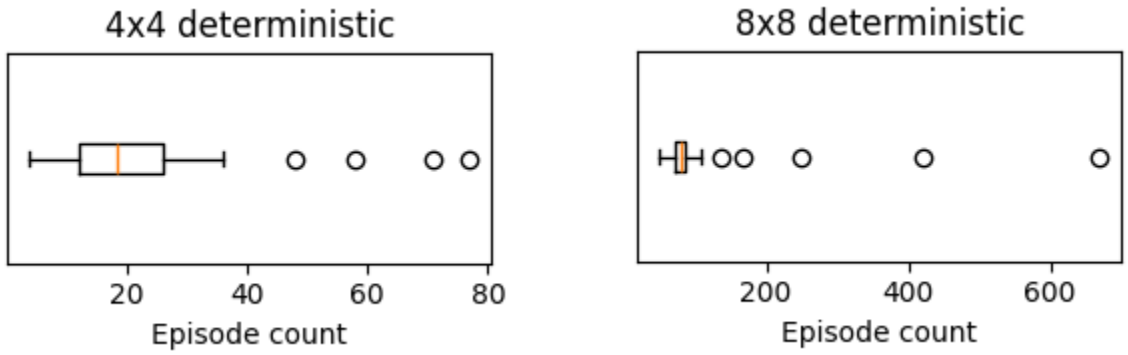
A) See attached `qlearn.py`. Check out the stuff at the bottom of the file for information on how to run the program with different environments and parameters. I got lazy and didn't make a CLI this time, sorry!

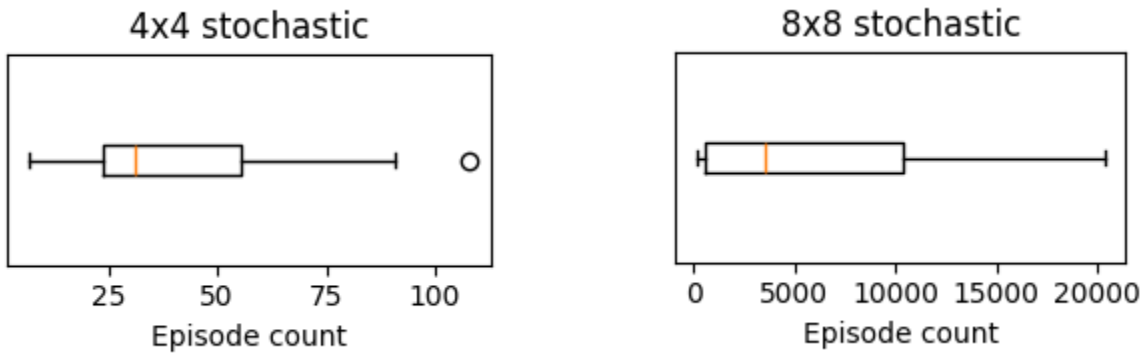
B) I implemented two different exploration strategies we discussed in class: `epsilon_greedy` and `state_counting`. I found that for slippery environments, `epsilon_greedy` with a random act probability of 0.1 works well, and for non-slippery environments, `state_counting` with $\alpha = 1$ (optimal for non-stochastic environments) works well. I arrived at these values after several hours of testing, debugging my implementation, testing again, more debugging, and a little crying of salty tears.

C) This data was collected based on 30 runs using a (`state_counting`, $k = 1$), $\alpha = 1$, $\gamma = 0.9$ agent with `policy_delta`, $n = 3$ convergence criteria; see E for the convergence criteria I implemented.



The 4x4 deterministic took an average of 23.0 episodes with a standard deviation of 18.3, while the 8x8 deterministic took an average of 119.0 episodes with a standard deviation of 124.0. Notably, the outliers are very far away from the main cluster of values for both run types.

D) This data was collected based on 30 runs using an (`epsilon_greedy`, $\epsilon = 0.1$), $\alpha = 0.05$, $\gamma = 0.9$ agent with `policy_delta`, $n = 3$ convergence criteria.



The 4x4 stochastic took an average of 40.3 episodes with a standard deviation of 24.5, while the 8x8 stochastic took an average of 6078.0 episodes with a standard deviation of 6291.5. `epsilon_greedy` seems to yield much more uniform results, although still right tailed. I noticed this pattern too while testing deterministic environments with `epsilon_greedy`; `state_counting` merely performed better, so I opted to use that exploration method.

E) I struggled a lot with finding a way to test for convergence, but settled on three different methods. It seems like in practical applications for Q-Learning, algorithms don't check for convergence, instead simply running for as long as possible; the `none` convergence criteria does this. I'm sure that's not really what this class is looking for, so I also implemented `policy_delta`, which checks if the policy hasn't changed for at least n episodes, and `v_delta`, which checks if the difference in average Q value per state between two consecutive runs is below some epsilon. I've found that `policy_delta` outperforms `v_delta` in converging quickly and without complaint, but `v_delta` is probably closer to what a formal convergence criteria definition should be.

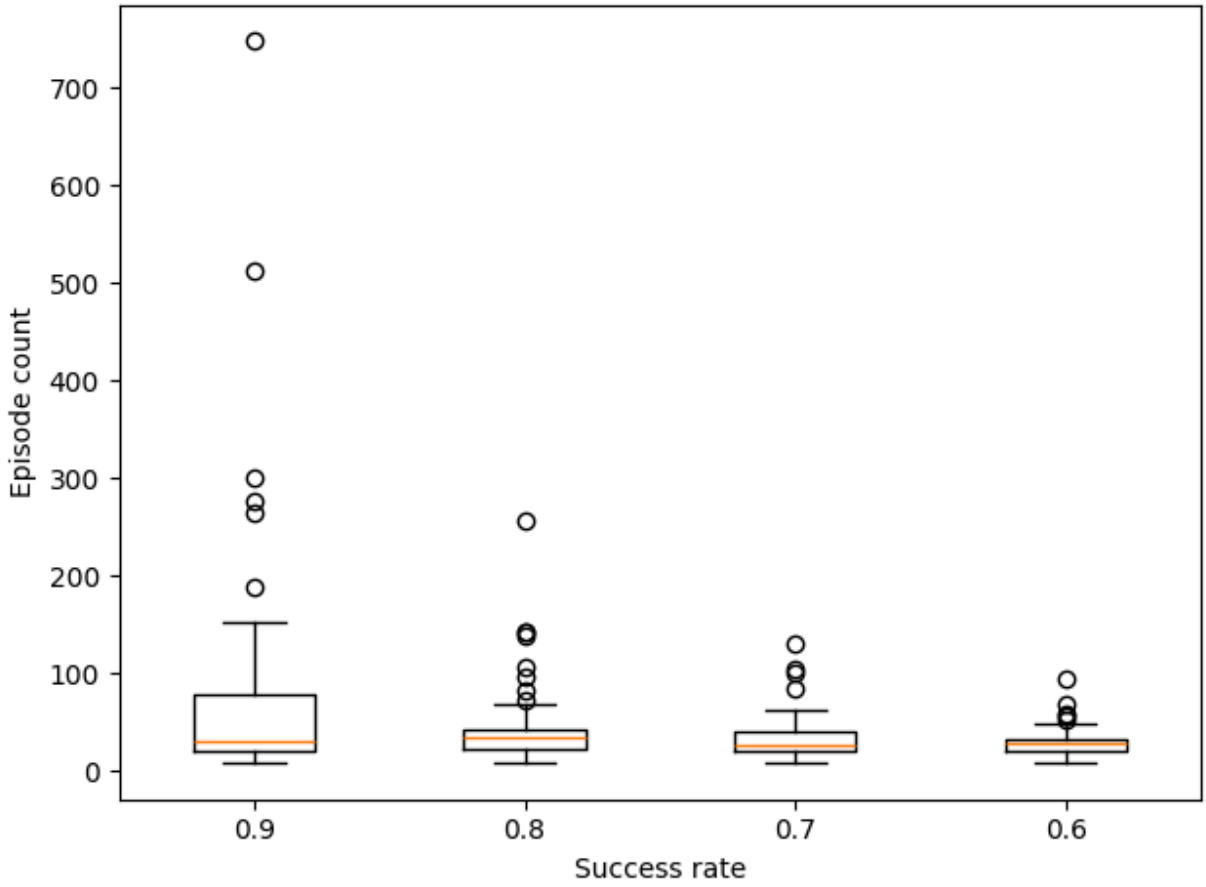
Fi) As mentioned above, `state_counting` is more suited for deterministic environments, while `epsilon_greedy` is more suited for stochastic environments.

	EG average	EG std	SC average	SC std
4x4 deterministic	173.7	291.6	23.0	18.3
8x8 deterministic	N/A	N/A	119.0	124.0
4x4 stochastic	40.3	24.5	20.7	13.6
8x8 stochastic	6078.0	6291.5	1143.3	2486.9

These are again the statistical results from 30 runs. `epsilon_greedy` never actually managed to converge for an 8x8 deterministic board. It would appear that the board is simply too large for semi-random movement to explore all the nooks and crannies; since some states are never reached by the agent the policy is forced to choose at random, leading to non-convergence.

During this data collection, `state_counting` actually worked quite well for the 4x4 stochastic environment. However, the 8x8 stochastic environment ran into my hard-coded 1,000,000 step limit several times, leading to the remarkably high standard deviation when compared to the average. This means `state_counting` wasn't able to converge given a million actions taken in the environment of the course of tens of thousands of episodes. Not good!

Fii) I ran my algorithm with standard (`epsilon_greedy`, 0.1) and (`policy_delta`, 3) parameters in a 4x4 environment with varying levels of success rate for 50 runs.



Surprisingly, it doesn't look like success rate impacts learning rate. If anything, increasing the success rate makes outliers more common! This is probably because larger success rates make it more difficult to reach all relevant states of an environment, making it harder for `policy_delta` to say a policy has converged.

Fiii) I ran my algorithm with standard (`epsilon_greedy`, 0.1) and (`policy_delta`, 3) parameters in a 4x4 0.75 environment with varying reward schedules.

	avg	std
(10, -10, 0)	39.5	24.8
(10, -1, 0)	30.3	19.9
(5, -5, 0)	43.0	25.0
(10, -10, -0.05)	49.5	18.0
(10, -1, -0.05)	74.4	46.0

Interesting results here, too. It looks like simply scaling the goal and hole rewards does little to affect how quickly we converge; I thought it might do something because of the gamma value, but I guess not. Making the goal reward larger in magnitude than the hole reward makes convergence a little easier, but adding a negative living reward makes it harder. It takes nearly twice as many episodes to converge with a negative living reward plus smaller magnitude hole reward; these two values conflict with each other, I suppose.