Journal: Assignment 2 - Reinforcement Learning

Kobee Raveendran

CAP5636 - Fall 2020

1 Q1: Value Iteration Agent

1.1 Initial Thoughts and Experimentation

I started with the value iteration agent implementation as it was part of Q1 on Berkeley's page. It also seemed the most intuitive to implement right off the bat. I started the project by first running GridWorld in manual mode, just to get a feel for the nature of MDPs, then ran the (very suboptimal) default agent through MazeGrid. Below is the result of the default agent's performance on MazeGrid.

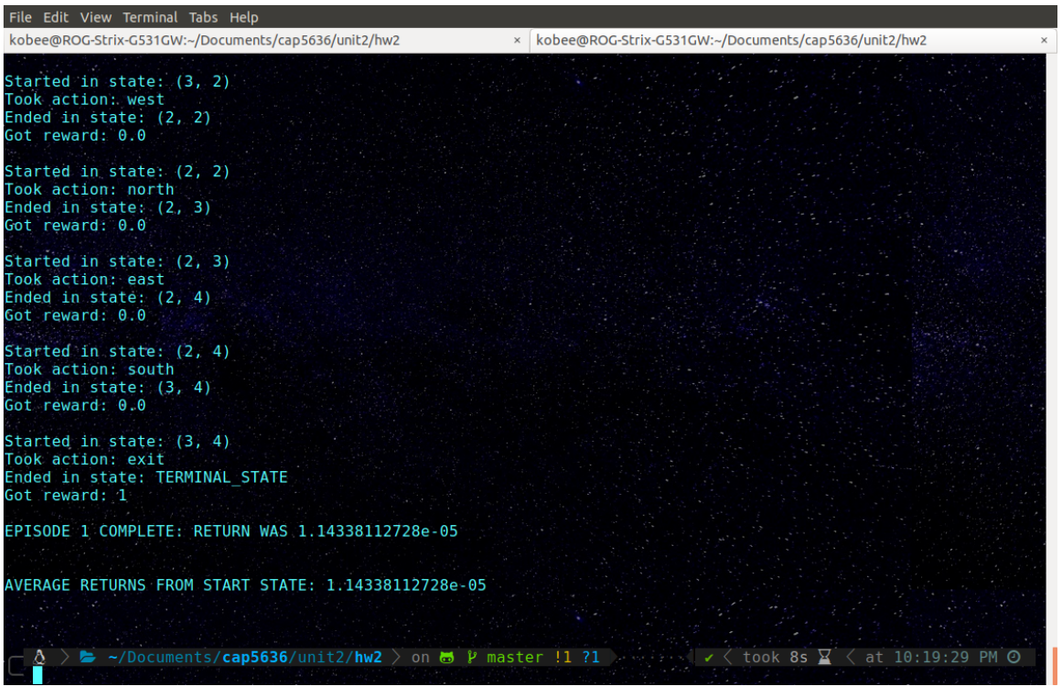
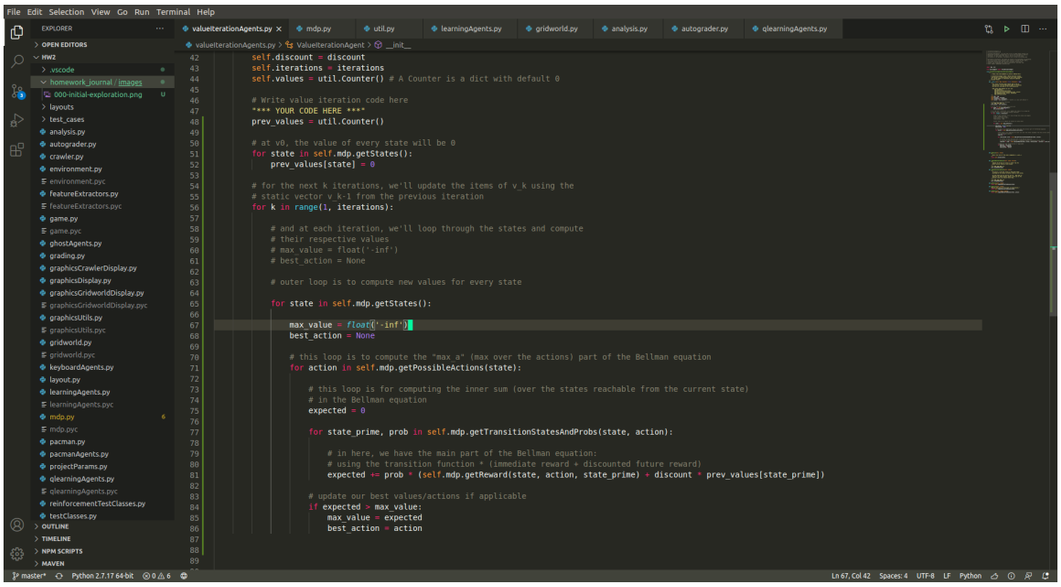


Figure 1: Default agent's performance on MazeGrid.

1.2 Initial Implementation

Much of what I based my initial implementation on was inspired by the UC Berkeley lectures on these topics. As such, most of my intuition on value iteration came from their lecture on MDPs. Dr. Klein's walkthrough of the algorithm in "programmatic terms" (i.e. using terminology like "loop over", "lookup", etc.) made it easier to translate the Bellman update equation into code, and what I ended up with based on that lecture is below. At this point, I haven't run it, but I think it's a good start.

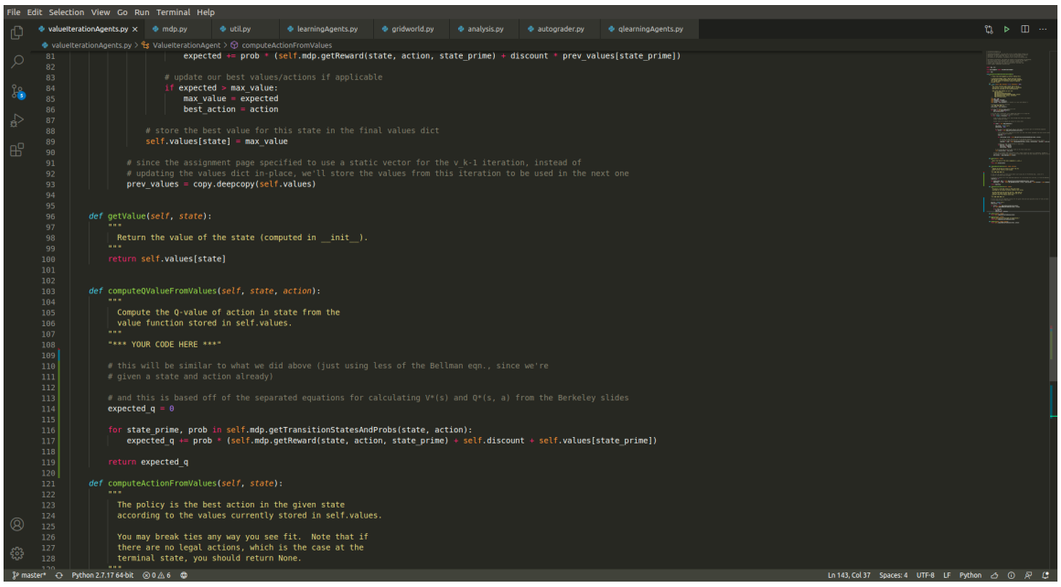


```
        self.discount = discount
        self.iterations = iterations
        self.values = util.Counter() # A Counter is a dict with default 0

        # Write value iteration code here
        "*** YOUR CODE HERE ***"
        prev_values = util.Counter()

        # at v0, the value of every state will be 0
        for state in self.mdp.getStates():
            prev_values[state] = 0

        # for the next k iterations, we'll update the items of v_k using the
        # static vector v_k-1 from the previous iteration
        for k in range(1, iterations):

            # and at each iteration, we'll loop through the states and compute
            # their respective values
            # max_value = float('-inf')
            # best_action = None

            # outer loop is to compute new values for every state

            for state in self.mdp.getStates():

                max_value = float('-inf')
                best_action = None

                # this loop is to compute the "max_a" (max over the actions) part of the Bellman equation
                for action in self.mdp.getPossibleActions(state):

                    # this loop is for computing the inner sum (over the states reachable from the current state)
                    # in the Bellman equation
                    expected = 0

                    for state_prime, prob in self.mdp.getTransitionStatesAndProbs(state, action):

                        # in here, we have the main part of the Bellman equation:
                        # using the transition function * (immediate reward + discounted future reward)
                        expected += prob * (self.mdp.getReward(state, action, state_prime) + discount * prev_values[state_prime])

                    # update our best values/actions if applicable
                    if expected > max_value:
                        max_value = expected
                        best_action = action
```

Figure 2: Initial value iteration implementation.

1.3 Updating States

Below is my code for both storing previous iteration state values, computing Q-values from the computed values, and computing the best action to take given a state. All of these were also based on the Berkeley lecture video on the topic.



```
                            expected += prob * (self.mdp.getReward(state, action, state_prime) + discount * prev_values[state_prime])

                        # update our best values/actions if applicable
                        if expected > max_value:
                            max_value = expected
                            best_action = action

                    # store the best value for this state in the final values dict
                    self.values[state] = max_value

                # since the assignment page specified to use a static vector for the v_k-1 iteration, instead of
                # updating the values dict in-place, we'll store the values from this iteration to be used in the next one
                prev_values = copy.deepcopy(self.values)

    def getValue(self, state):
        """
          Return the value of the state (computed in __init__).
        """
        return self.values[state]


    def computeQValueFromValues(self, state, action):
        """
          Compute the Q-value of action in state from the
          value function stored in self.values.
        """
        "*** YOUR CODE HERE ***"

        # this will be similar to what we did above (just using less of the Bellman eqn., since we're
        # given a state and action already)

        # and this is based off of the separated equations for calculating V*(s) and Q*(s, a) from the Berkeley slides
        expected_q = 0

        for state_prime, prob in self.mdp.getTransitionStatesAndProbs(state, action):
            expected_q += prob * (self.mdp.getReward(state, action, state_prime) + self.discount + self.values[state_prime])

        return expected_q

    def computeActionFromValues(self, state):
        """
          The policy is the best action in the given state
          according to the values currently stored in self.values.

          You may break ties any way you see fit.  Note that if
          there are no legal actions, which is the case at the
          terminal state, you should return None.
        """
```

Figure 3: Initial implementation of Q-value calculation from values.

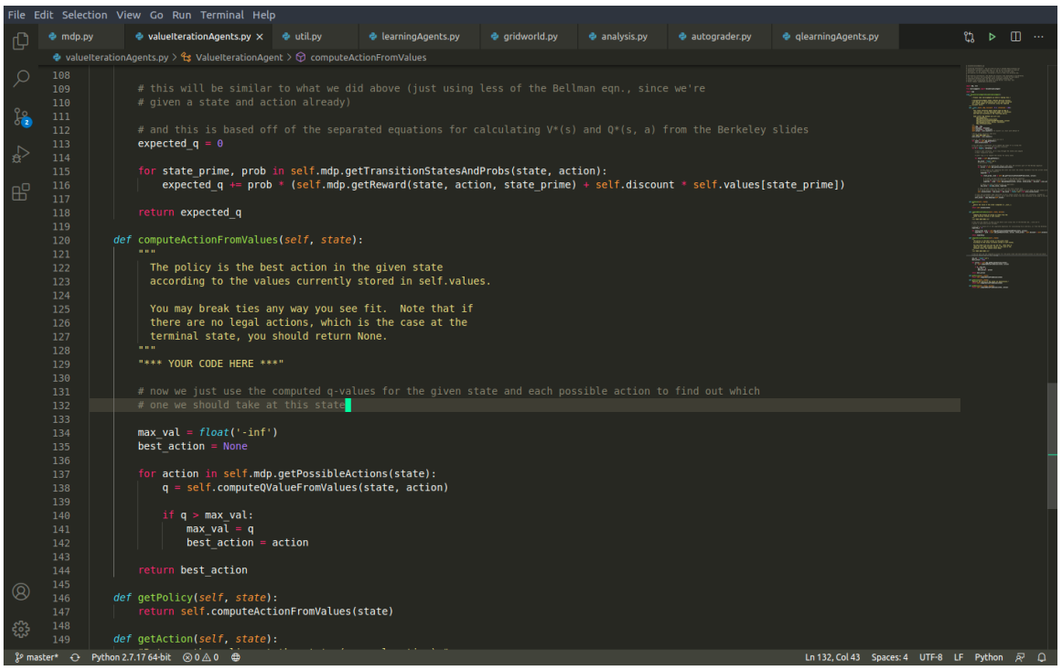


Figure 4: Initial implementation of optimal action calculation from values.

1.4 Testing Value Iteration

1.5 Sprint 1

At this point I wanted to test my code, and unfortunately was met with a pretty big roadblock: none of the autograder's tests passed.

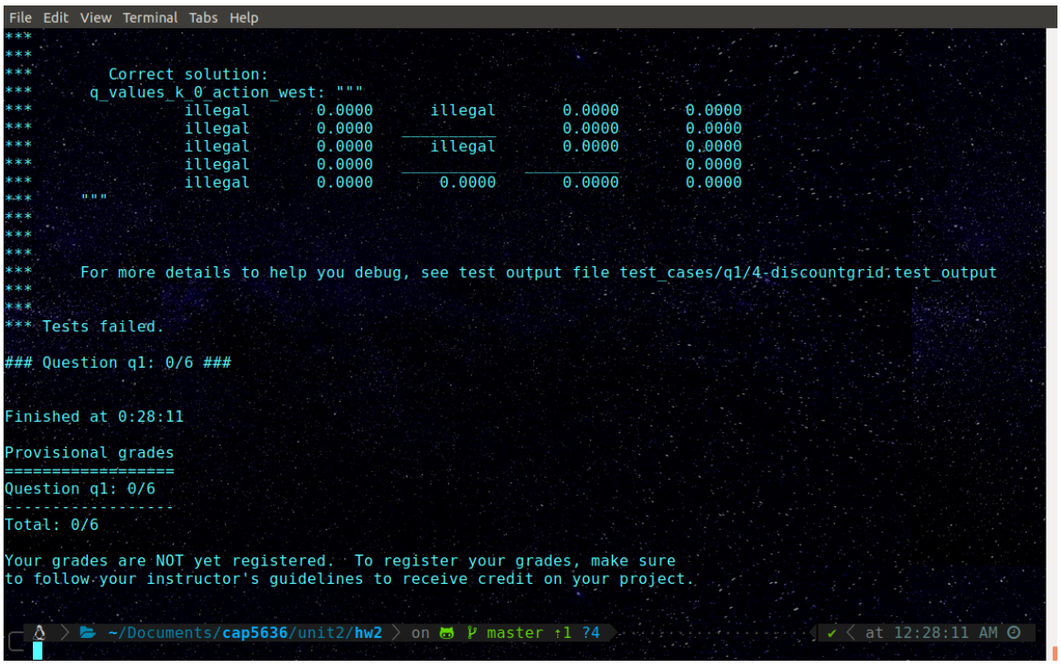


Figure 5: Initial test results from running Value Iteration from Q1 on GridWorld.

Additionally, after attempting to run my code in the "interactive"/visual mode (with the GUI), I was met with some errors.

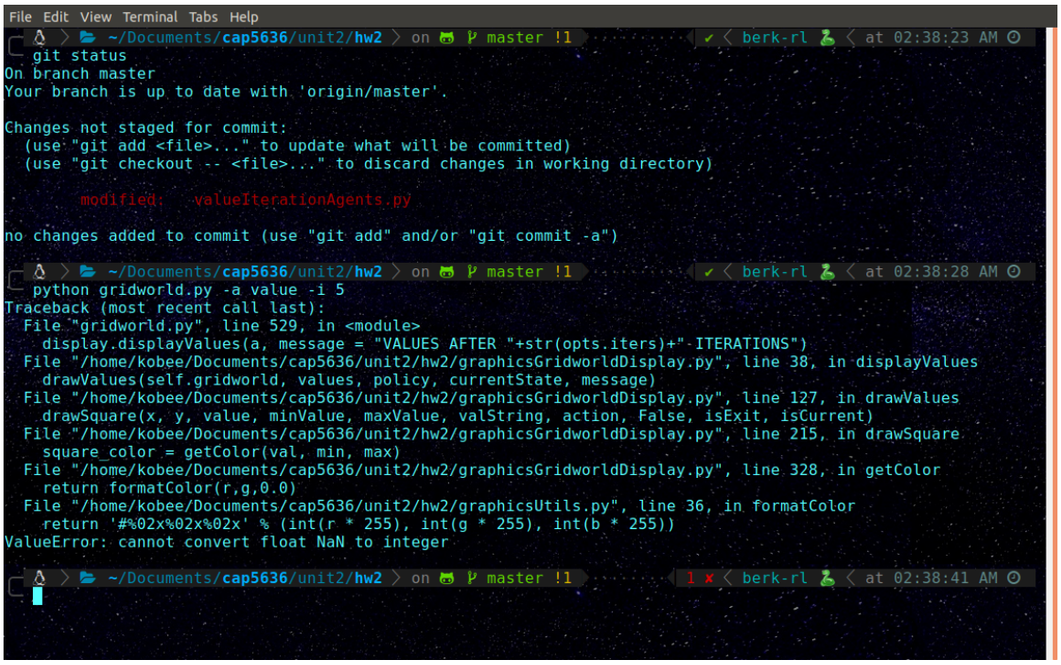


Figure 6: Errors from running my initial implementation on the GUI.

I eventually narrowed these errors down to issues that my current code had, which were related to how I was handling **terminal states**. According to the description in the `mdp` class definition, terminal states may not even have an action space. This meant that my action loop would not execute, and I'd return the original `max_value`, which would be a float representing negative infinity (which likely caused the errors). After inserting a check to see that the max value was not infinity, my code (below) finally ran without errors. If a terminal state was encountered, I'd simply give it the value it had in the previous state (simulating a no-loop), instead of giving it negative infinity like it had previously.

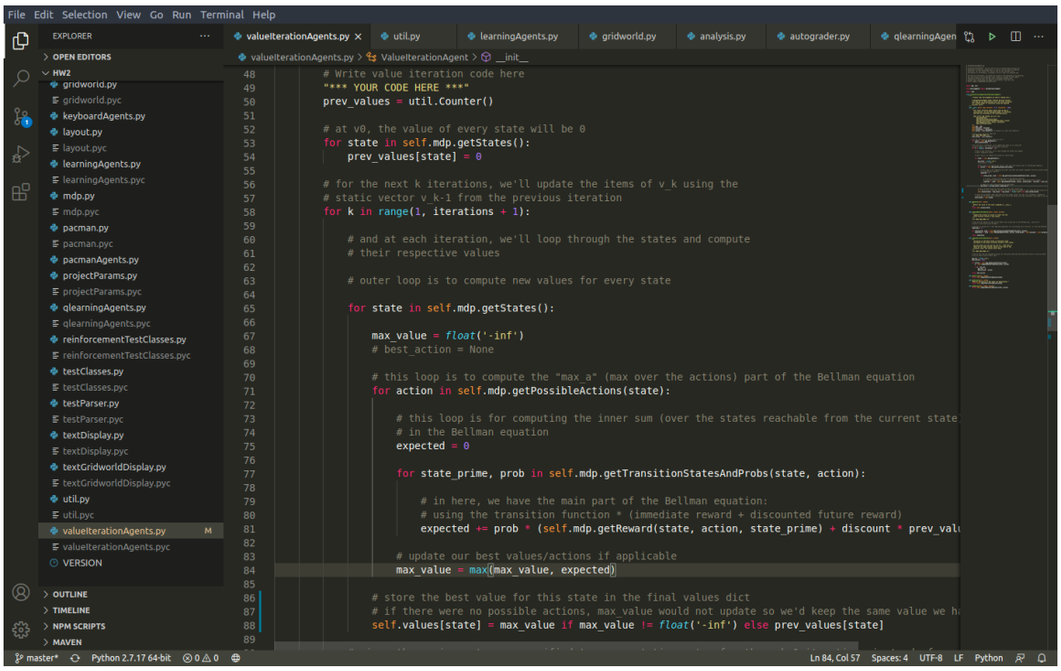


Figure 7: Error-free code for value iteration, achieved by correctly handling terminal states.

1.6 Successful Value Iteration Runs

At this point, my code actual run without errors in the GUI. However, all autograder tests were still failing, which meant that I was computing values wrong for each iteration. To find out what was going on, I checked my 5-iteration results with the GUI demo against the screenshot provided on Berkeley's webpage. I found that I was getting results very close to, but slightly off from, the results shown on the webpage.

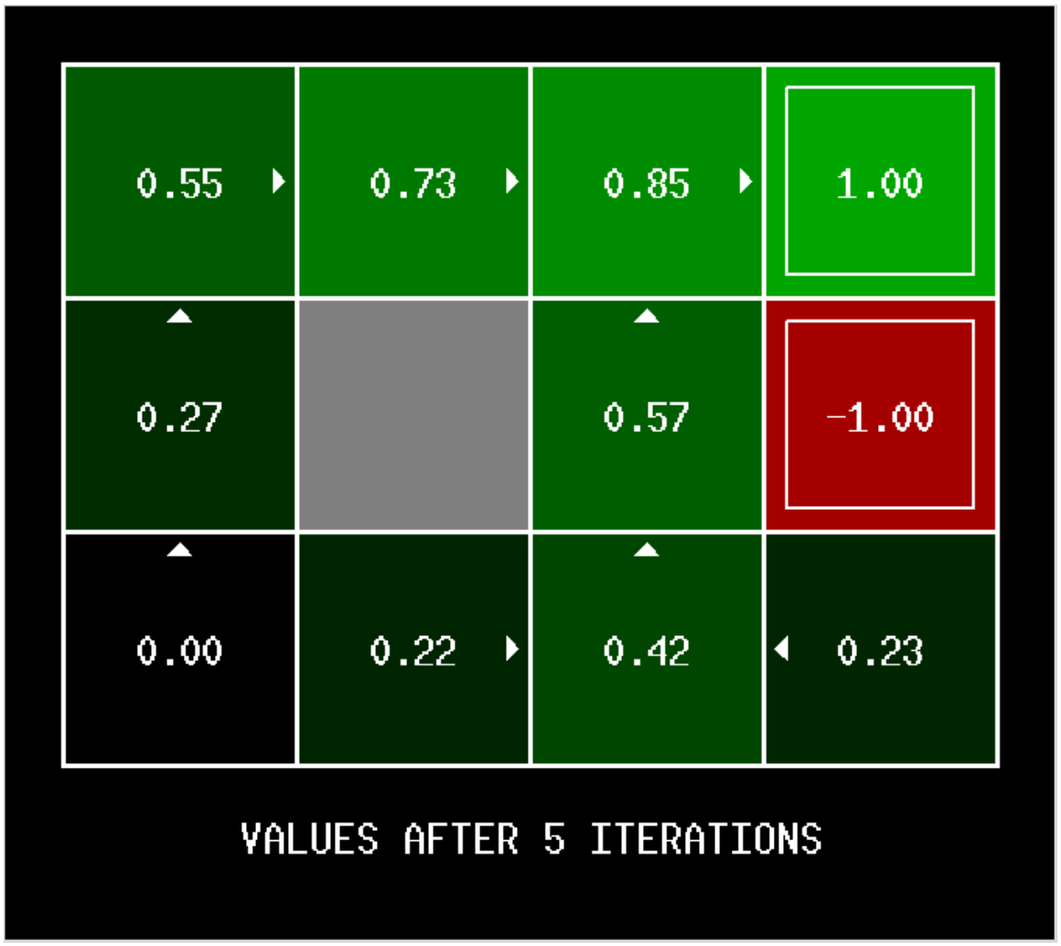


Figure 8: Initial results of value iteration over 5 iterations. Note that the values for each cell vary ever so slightly from the expected results on the Berkeley webpage.

I was confident that the logic of the implementation was correct, and after a long time narrowed it down to a programming error. I found it by comparing my testing something in my current code: in-place value updates. That is, instead of using a `prev_values` dict, I just used `self.values`, and found that I got the same results. This meant that my dict was *not* correctly capturing a snapshot of the previous values. From then, I quickly realized that I had passed a reference of the current values dict in my "assignment" step at the end of each iteration, rather than copying all the values into a new dictionary. After fixing this with a `deepcopy` of the current iteration's values, the results finally matched up correctly with Berkeley's.

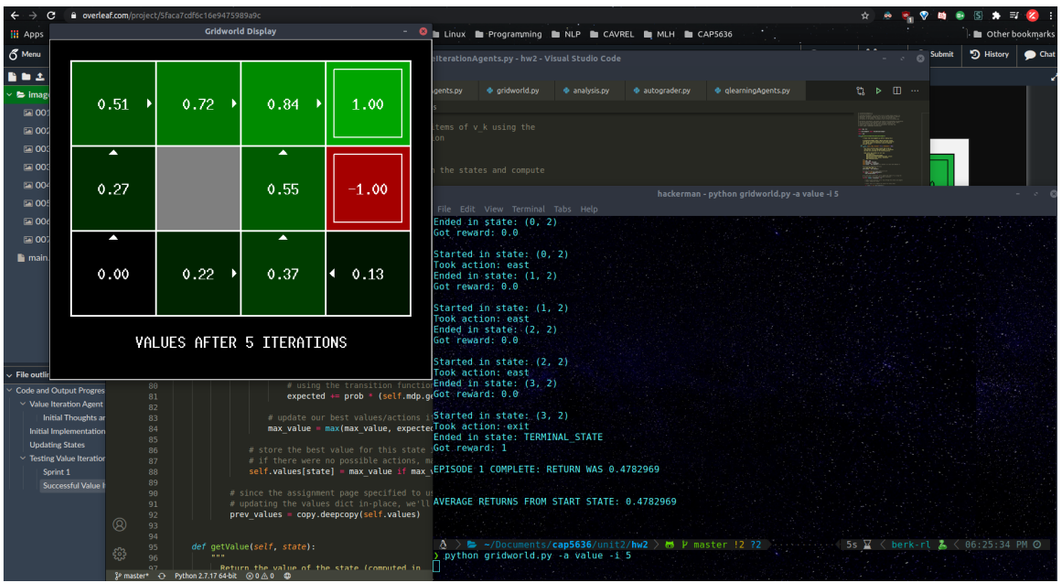


Figure 9: New results on the 5-iteration test after fixing previous value assignment.

After fixing this, I tested again using the autograder, and now all tests passed for Q1!

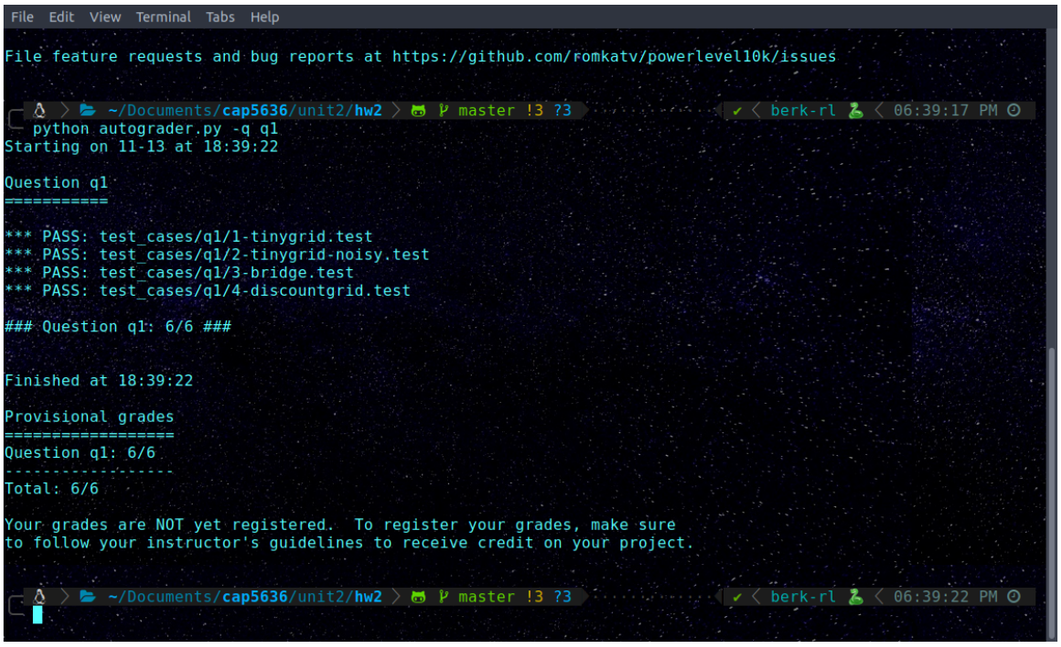


Figure 10: Results from running the autograder on Q1 with my updated value iteration code.

2 Q2: Bridge Crossing Analysis

This section of the assignment was much simpler because it only involved tweaking parameters, and only tested our intuition of each parameter's effect on agent actions. Thus, it was very experimental and heavily based on trial and error. The goal is to get the agent to avoid taking the immediate, low-value reward and instead attempt to cross the bridge to get the larger reward. And since we can only change one of either the discount or noise parameters, it had to be one that had a high impact on the agent's current policy.

Intuitively, the only reason an agent would *not* want to cross the bridge is to avoid landing in the fire pits. An agent acting optimally will *only* land in a fire pit if it "slips" due to randomness in the transition function, not because it actively wanted to end up in that state. So, to help the agent move along and "eliminate" its fear, I just reduced the randomness of its actions (the noise) until the agent could be confident enough to consistently cross the bridge. At this point, the noise would be so low that the agent can move with almost absolute certainty. Keeping discount at 0.9, I constantly halved the noise value until the agent finally crossed; the value of noise that I ended up with was 0.012 (though lower noise values would also work, since the agent would fear crossing the bridge even less).

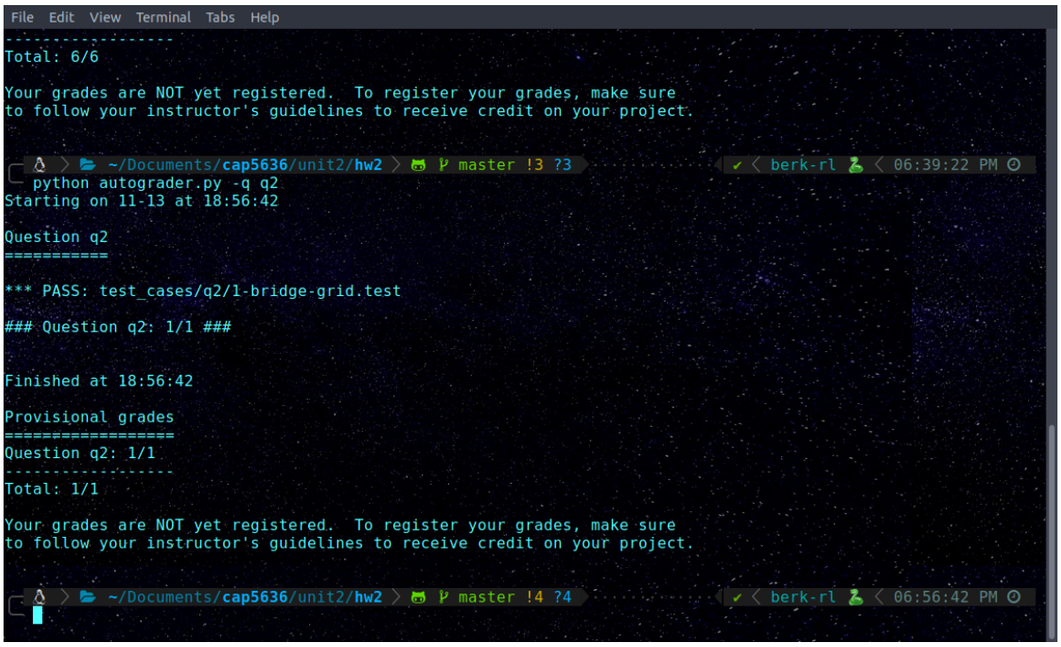


Figure 11: Autograder results from changing the environment parameters as described above.

Figure 12: New values after using the new parameter updates. The values along the bridge are finally positive, meaning the agent should now attempt to cross the bridge.

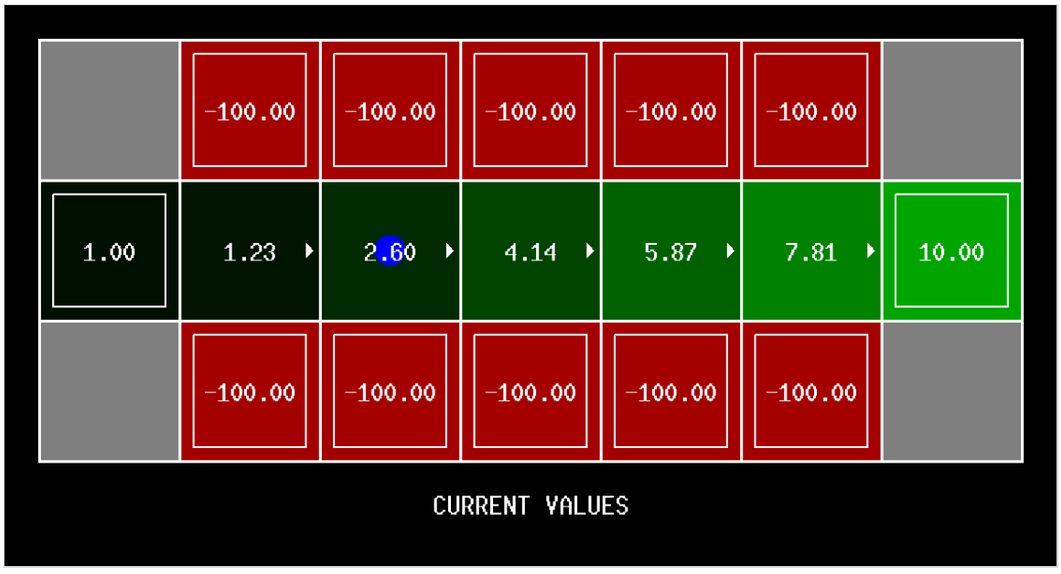


Figure 13: Agent partway through the BridgeCrossing demo. Using the new parameters (`noise = 0.012`, we see that it finally musters the courage to cross the bridge.

3 Q3: Policies

Like the previous section, this question also tested our intuition of the factors affecting an agent's policy. But this time, there is much more flexibility in how an agent can go about reaching a goal, and the parameters we set should reflect that. Again, this section is much more about trial and error backed by intuition.

3.1 A: Prefer the nearby exit, risking the cliff

For this problem, I drew partially from Q2 in that I wanted to reduce the agent's fear of the cliff (since the agent *should risk the cliff* as part of the problem). To do this, I once again iteratively decreased the noise to the point where the agent would gain the confidence to walk along the cliff.

To prefer the nearby exit, however, noise would not be enough. If I had only changed the noise, the agent would instead rather go around the long way, and probably also go to the other goal, since it has a higher reward. So, to steer the agent toward the closer goal, I heavily discounted future rewards (as the other goal is farther, and would thus be reached later). In addition, I also incorporated a negative living reward to additionally push the agent to avoid going the long way and rush to the nearest exit.

My updated parameters are a discount factor of 0.2, a noise value of 0.0025, and a living reward of -2.

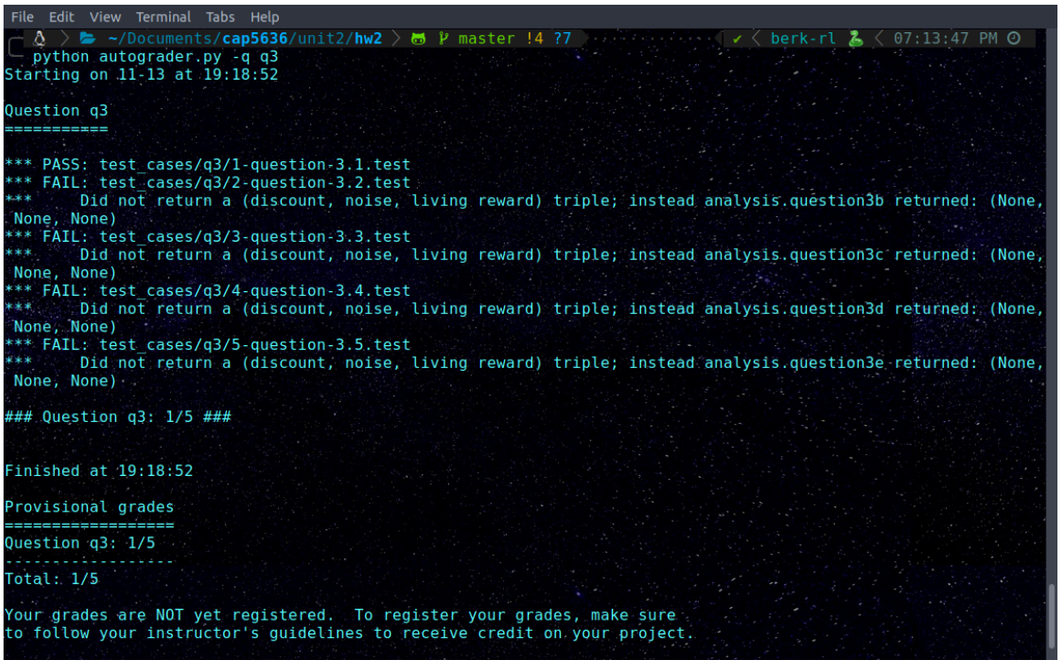


Figure 14: Autograder results for Q3, in which subproblem A passes.

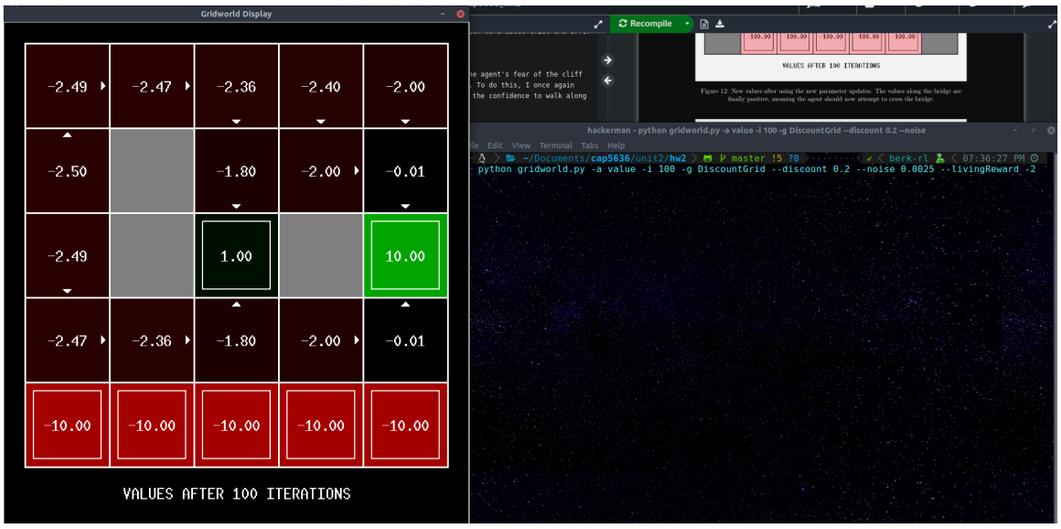


Figure 15: Values from the DiscountGrid GUI for the parameters applied in subproblem A. We see that the optimal policy from the start state (0, 1) is to go east, then north into the nearby goal with reward 1.

3.2 B: Prefer the nearby exit, avoiding the cliff

This time around, we want the agent to avoid the cliff but still prioritize the nearby exit. So, similar to part A, I do this by heavily discounting future rewards, making the agent want to go for better immediate rewards. However, to avoid risking the cliff, I raise the randomness (noise) substantially, instilling the fear of slipping back into the agent. And finally, to prevent the agent from risking the cliff, I also make the living reward 0 (instead of -2) so that it can take its time and go the long way around. If I had instead used a positive living reward, the agent would not want to exit early, and may instead choose to follow a cycle or take longer and go for the exit with a higher reward.

My updated parameters are a discount factor of 0.2, a noise value of 0.2, and a living reward of 0.

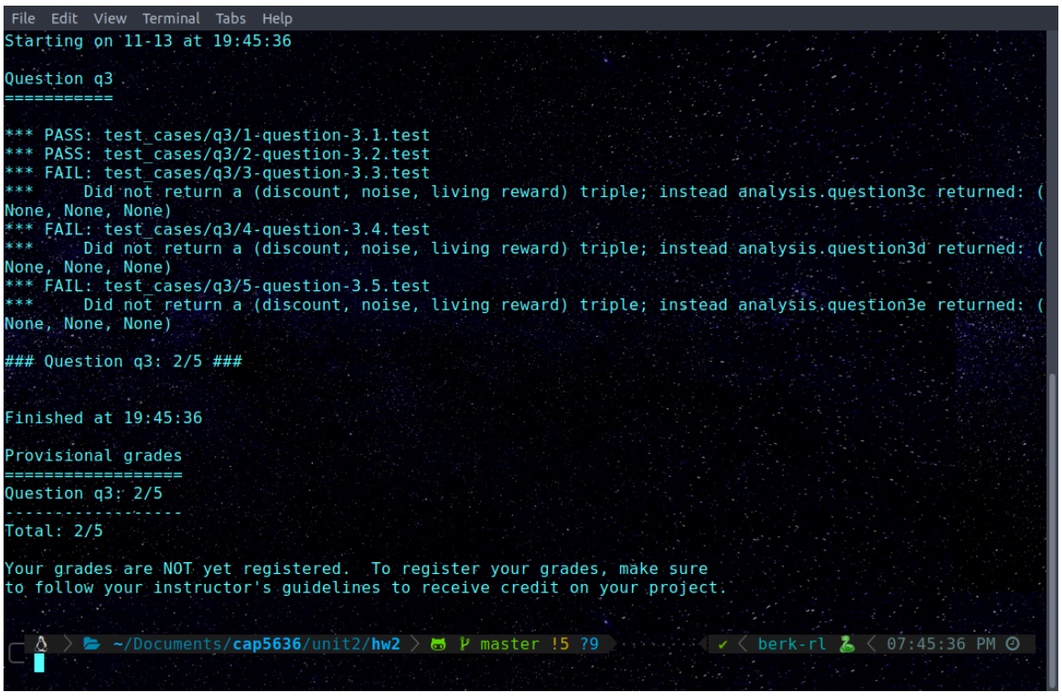


Figure 16: Autograder results for Q3, in which subproblem B also passes.

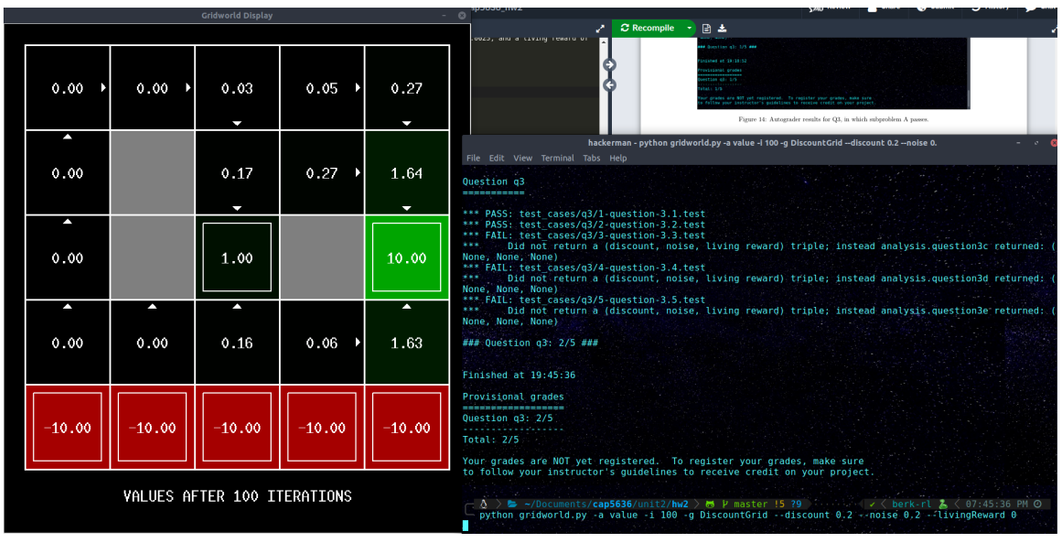


Figure 17: Values from the DiscountGrid GUI for parameters applied in subproblem B. We see that the optimal policy from the start state is to go north, then east at the top left corner, then south in the column containing the nearby goal state.

3.3 C: Prefer the distant exit, risking the cliff

Now, we want to reach the *farther* exit, but while risking the cliff. In a similar fashion to Q3A, I use a very low noise level to remove the agent's fear of slipping off the cliff. This time, however, we don't want to agent to prioritize the early exit, so I raised the discount factor to 1, so future rewards are not discounted at all. This gives the agent freedom to choose the highest reward on the map, which is the further exit in DiscountGrid. And to additionally make the agent avoid taking the long way around, I introduce the same negative living reward from Q3A.

My updated parameters are a discount factor of 1.0, a noise value of 0.0025, and a living reward of -2.

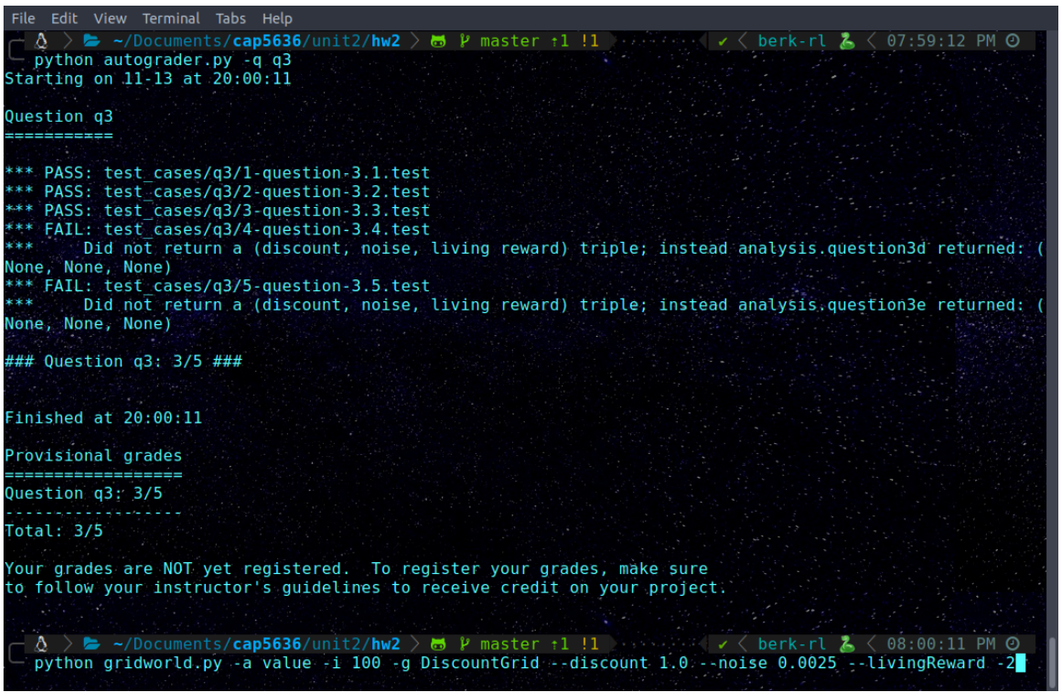


Figure 18: Autograder results for Q3, in which subproblem C also passes.

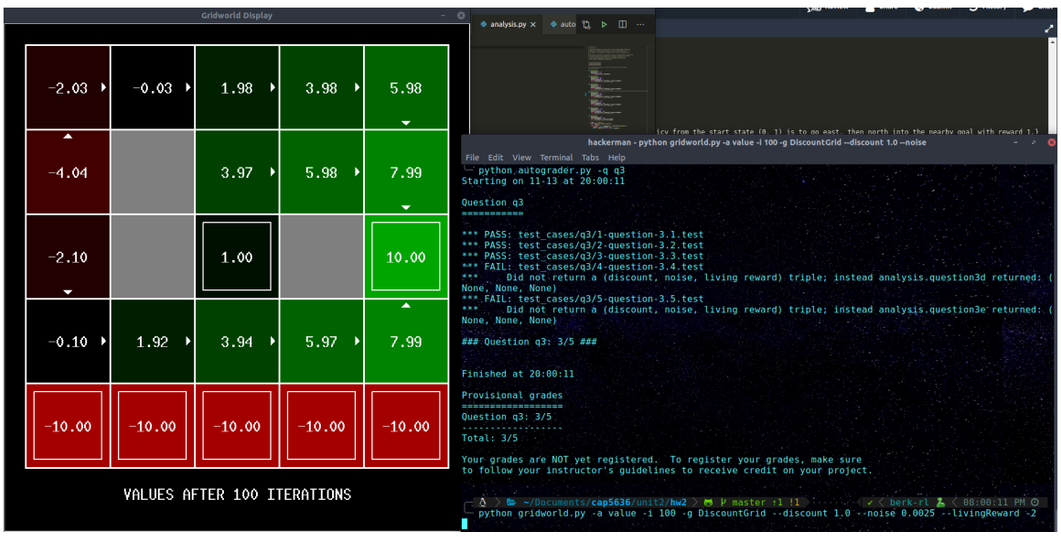


Figure 19: Values from the DiscountGrid GUI for parameters applied in subproblem C. We see that the optimal policy from the start state is to go right, risking the cliff, but *skipping* the nearby exit and instead going to the more distant exit with higher reward.

3.4 D: Prefer the distant exit, avoiding the cliff

In this case, we want to avoid the cliff, so consistent with the previous case, I increase the noise until the the risk is sufficiently large for the agent to want to avoid the cliff. Additionally, similar to Q3C, we also want the agent to go for the distant exit by not discounting future rewards. And finally, to have it go the "long way" around, we make the living reward 0. This in tandem with the high randomness from the noise will encourage the agent to take the safe route, since it can do so without losing anything.

My updated parameters are a discount factor of 1.0, a noise value of 0.4, and a living reward of 0.

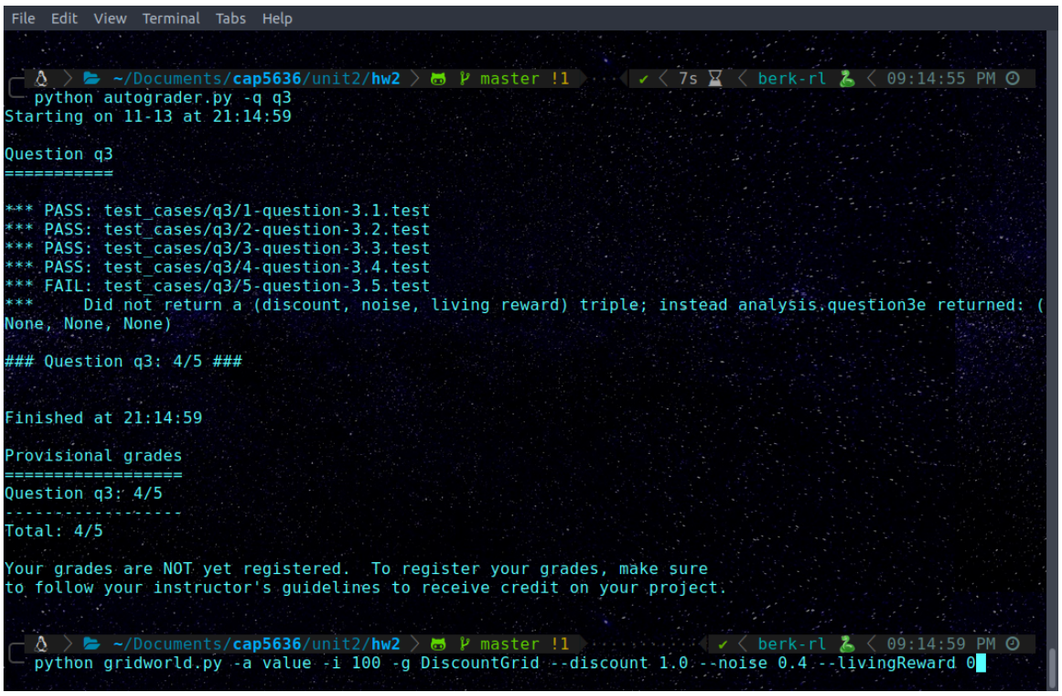


Figure 20: Autograder results for Q3, in which subproblem D also passes.

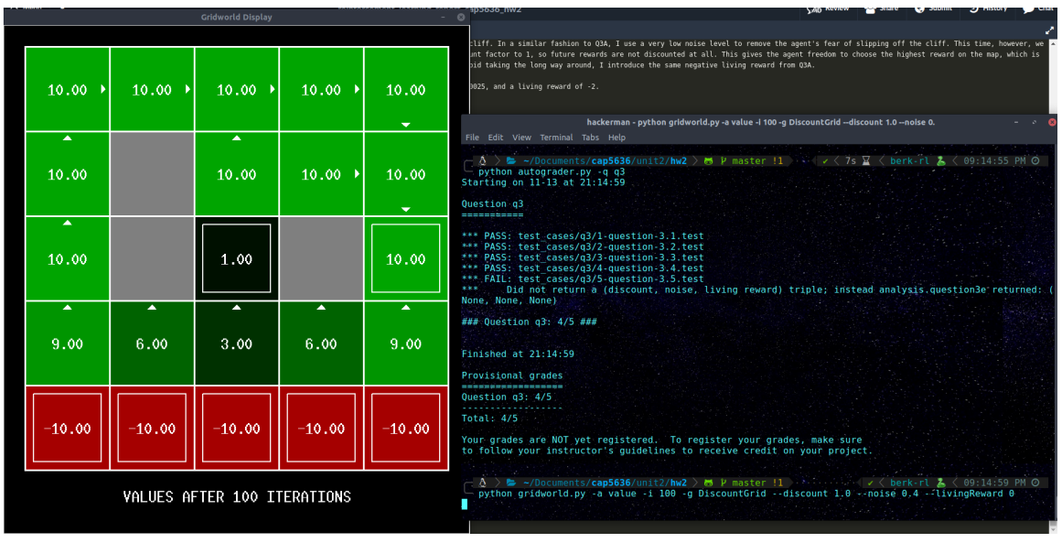


Figure 21: Values from the DiscountGrid GUI for parameters applied in subproblem C. We see that the optimal policy from the start state is to go north, avoiding the cliff, then going east and down to avoid the nearby exit and prioritize the distant one. Note that, even if the agent "slips" and goes to a cell closer to the nearby exit, the optimal action is always oriented in a way such that any transition positions the agent away from the nearby goal. However, one downside of my parameters is that if the agent *happens* to slip early on and toes the cliff line, it will likely take the earlier exit. But this would not be a result of the agent's conscious decision.

3.5 E: Avoid both exits and the cliff

Finally, in this case, we want to avoid even reaching a goal state while also avoiding a failure state. Intuitively, an agent would only want to avoid a goal state if the reward it gets from wandering around exceeds the possible rewards of pursuing a goal state (as I discussed above). So, to make this happen, I simply provide a positive living reward, and to prevent the agent from wanting to end the run, I do not discount future rewards. To avoid the cliff, I initially made the noise somewhat high to induce "fear" in the agent. However, after running the demo several times, I noticed the agent would often accidentally fall into the pits or reach a goal state due to the randomness (though it still passes the autograder since the policy avoids the goals and cliff anyway). So to improve on this, since the agent will not want to end anyway, I would rather give the agent full control. Since the negative reward of the cliff will always be lower than the cumulative living rewards, giving the agent full control will allow it to avoid the bottom part of the map entirely, and loop reward cycles by running into walls. In this way, all optimal actions on the line toeing the cliff will point away from the cliff (and away from the goal states), and the agent will be guaranteed to not fall into the

cliff.

My updated parameters are a discount factor of 1.0, a noise value of 0.0, and a living reward of 1.

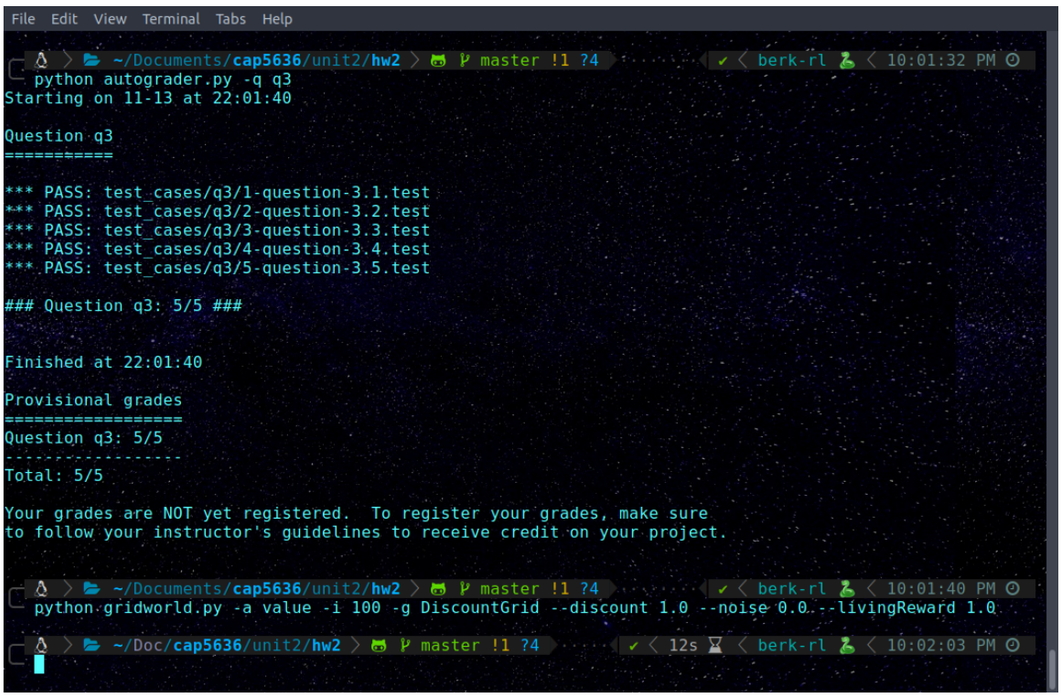


Figure 22: Autograder results for Q3, in which subproblem E also passes.

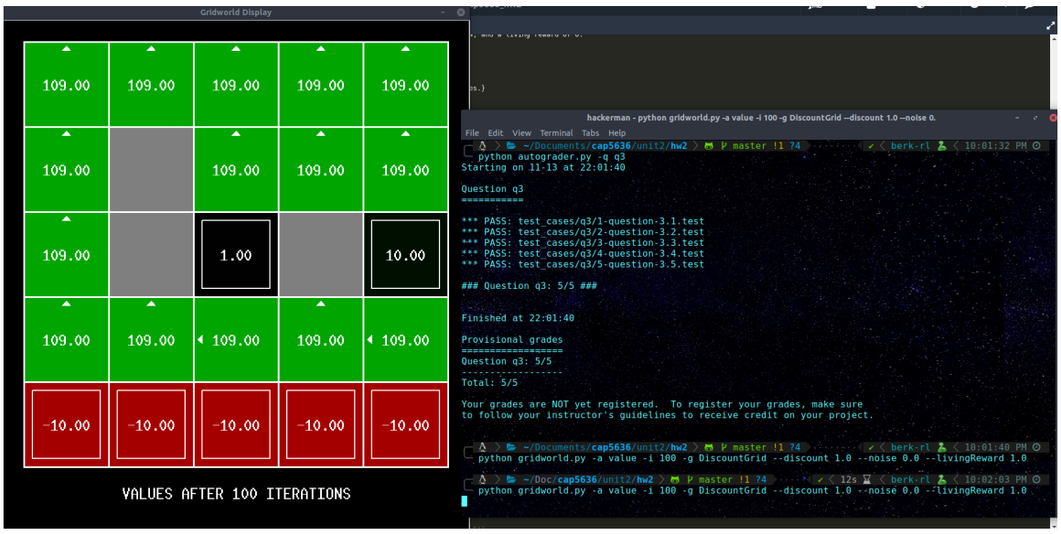


Figure 23: Values from the DiscountGrid GUI for parameters applied in subproblem C. We see that from the start state, all optimal actions tend to point away from both cliff edges *and* goal states. Many cells even make the agent run into walls to make the agent gain continual rewards.

4 Q4: Q-Learning

4.1 Q-Value and Action Computation

With Q-learning, rather than following pre-determined instructions from a policy, the task is left to the agent to explore and learn a good policy from almost nothing but its own experiences. However, the method of choosing good actions still boils down to choosing an action that yields the best reward (in this case, we maximize the Q-value). With that in mind, I started filling out some of the helper functions in `qlearningAgents` using the hints given in the document and formulas from the Berkeley RL lecture. Below is my initial attempt at the first three pieces: initializing the Q-values counter dictionary, retrieving Q values from the dictionary, computing the best value given a state, and computing the best action given a state (using just Q-values this time). Getting Q-values is slightly different from just values, since there is a Q-value for every action within every state, so I decided to handle this by making every key in the dict a tuple of (state, action). Computing the

best Q-value and action in the next two functions was fairly straightforward and consistent with the descriptions in the file.

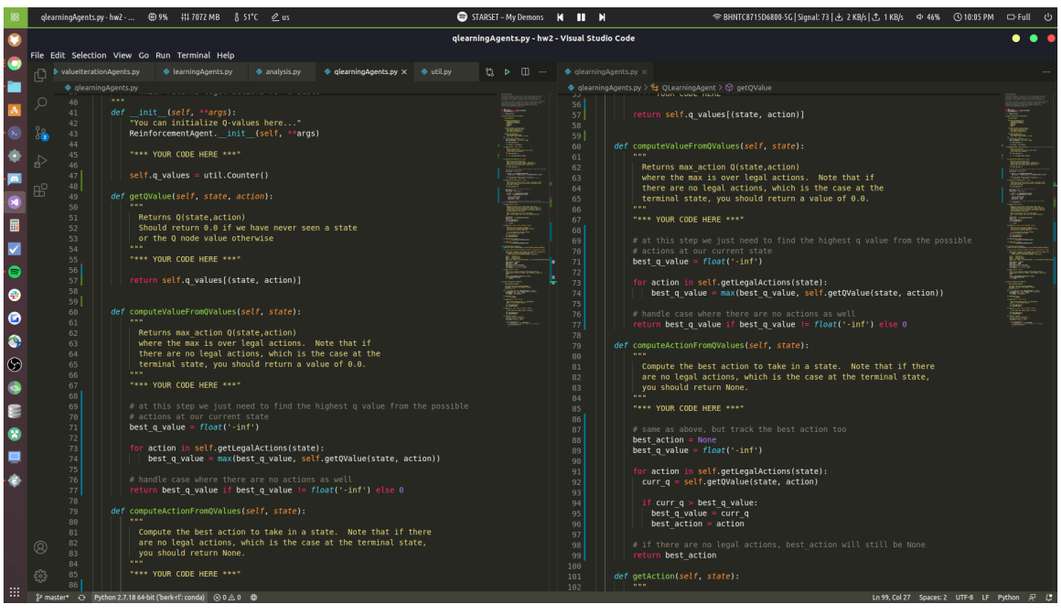


Figure 24: Retrieving Q-values, computing the best Q-value at a state, and computing the best action at a state.

4.2 Exploration vs. Optimal Action and Update Rule

At this point, I also filled in `getAction()` and `update()`. For `getAction()`, it became a simple matter of first choosing whether to take the optimal action or explore some random action (if actions were available), then executing on that decision using the previous functions. For the update rule, I first compute the sample, which considers the immediate reward and optimal future discounted rewards, then factors in learning rate and the previous experiences of our agent (from `self.q_values`) to update the current (state, action) pair with its newfound knowledge from exploring one step ahead.

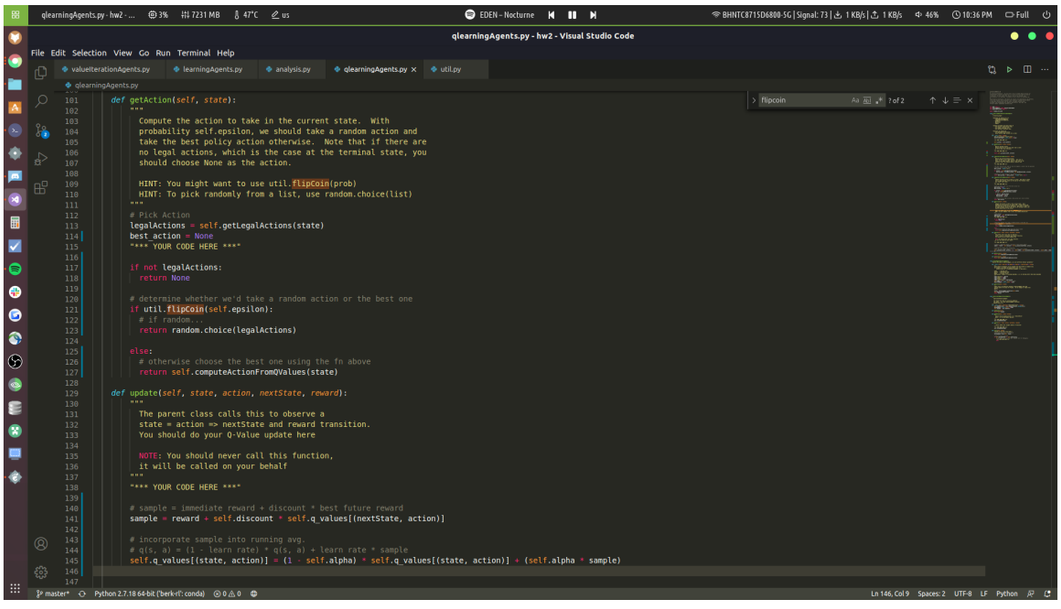


Figure 25: How the agent decides to choose an optimal action at a state or explore a random option. The update rule encodes the agent's new knowledge gained from taking an action.

4.3 Initial Run

To test out my code, I ran the demo as specified in the writeup. I noticed that, after doing 4 episodes of Q-learning as the writeup explained, I *did* get the correct Q-value for the terminal state, but noticed that the other states were not updating.

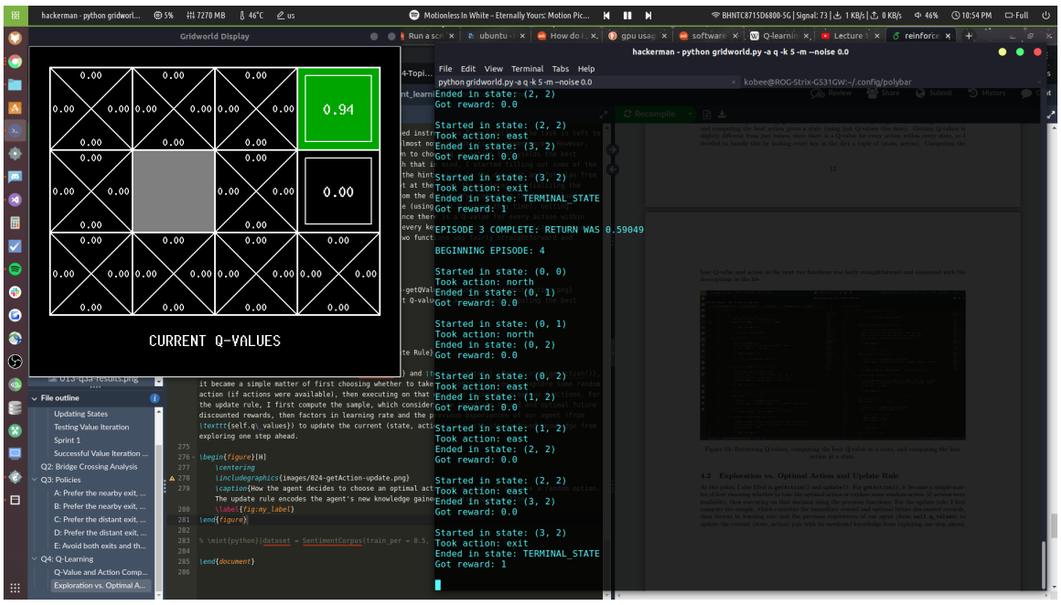


Figure 26: Initial demo run on GridWorld. Terminal state values update correctly, but the "learning wake" remains zero.

So, clearly, my Q-values were being computed just fine, but all the non-terminal states were not updating properly. After staring at the update function for a while, and comparing it to the equation (and explanation) from the lecture, I found my problem: I was using the current Q-values of (nextState, action) pair instead of the value that'd be stored in (nextState, nextAction) - the result of taking the *best action* in the next state. What I had previously only gave the current Q-value of taking the *current* action at the next state. So, I had to instead use my Q-value computation function to find the best q-value among all actions at the *next state*, rather than using the value in my current Q-values dict.

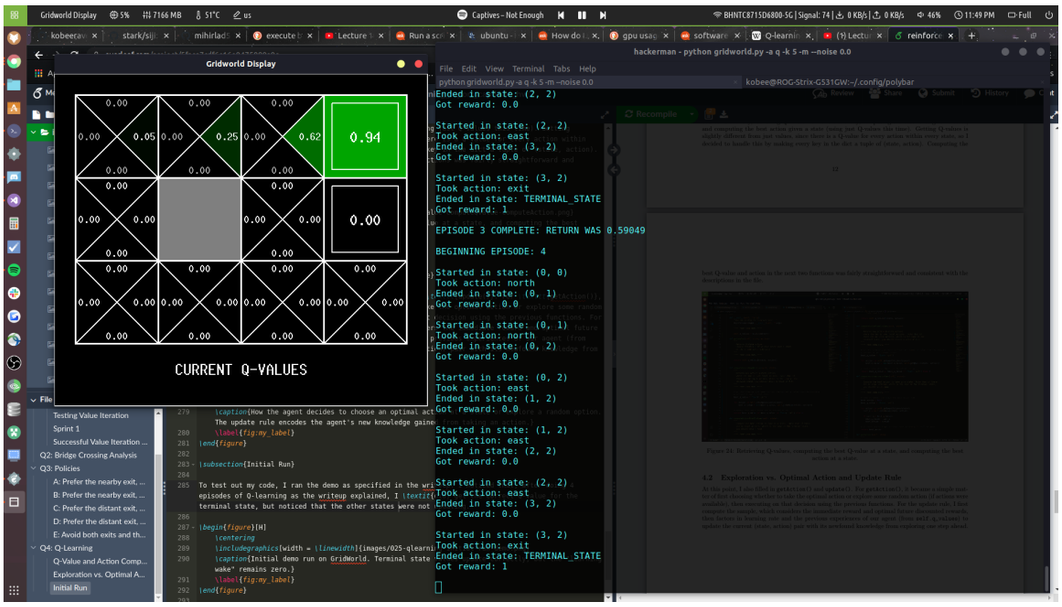


Figure 27: Another run of the Gridworld demo after fixing my update function. With future values now correctly computed, the Q-values after 4 episodes match the ones in the writeup.

To make sure my implementation worked in general, I ran it through the autograder. This time around, it passed all test cases!

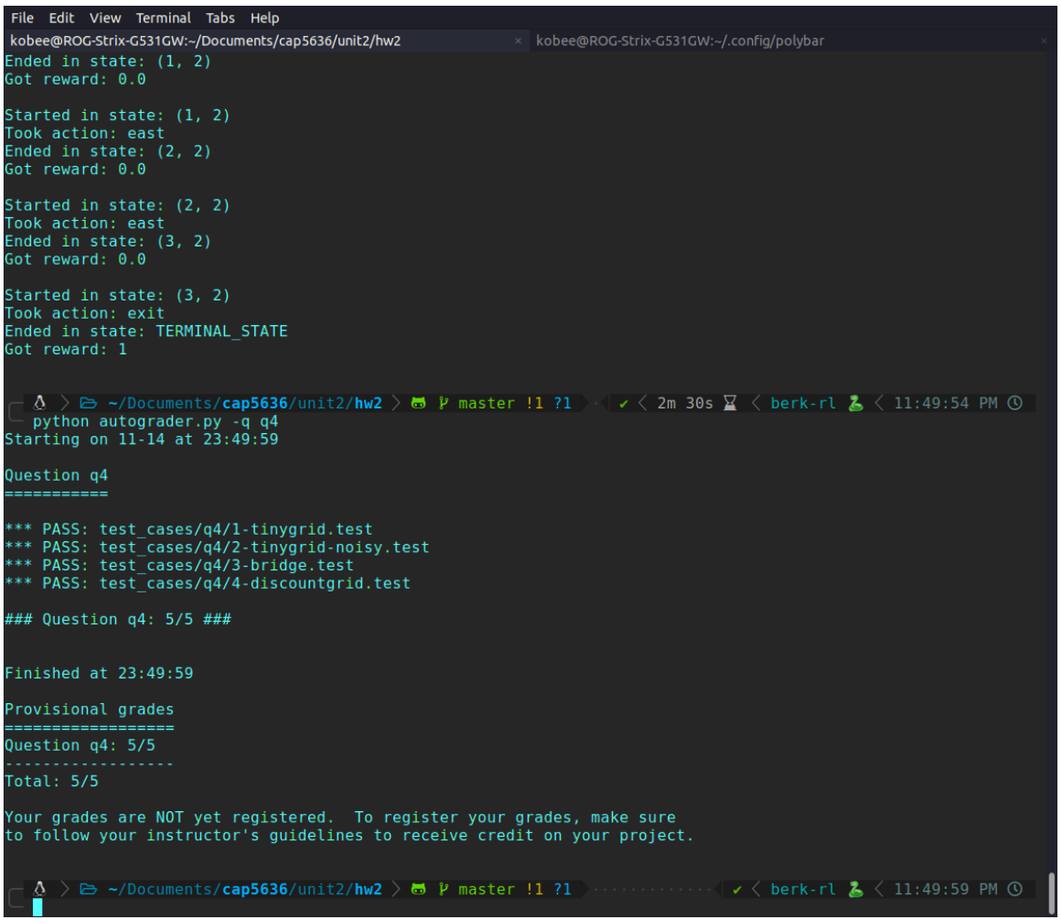


Figure 28: Results of my updated implementation of Q4 on the autograder, passing all test cases.

And for reference, here's the new implementation of the `update()` function that allowed my agent to learn properly.

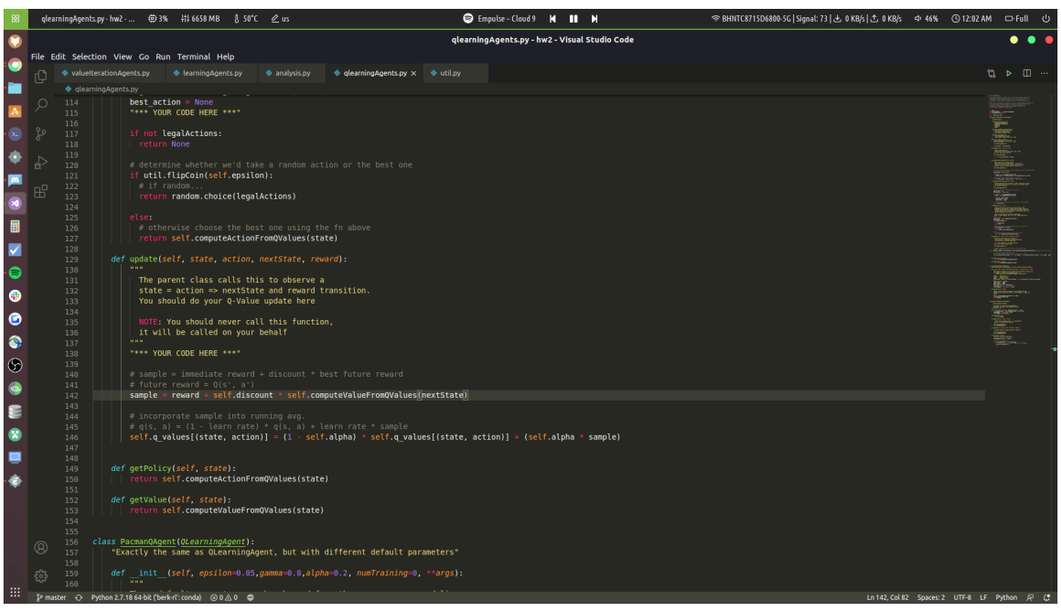


Figure 29: New update function. Now, it properly calculates the best value attainable from a one-step lookahead, rather than taking the best value that would be in the next state at the current timestep.

5 Value Iteration Visualization

To generate plots for these steps (steps 2 and 3), I used `matplotlib`. For value iteration, I generated plots for `iterations = 5` and `iterations = 100`, and for Q-learning I generated plots for `episodes = 5` and `episodes = 100`.

5.1 Value Iteration Plots

These plots were fairly simple to generate, as I could maintain a list of value updates throughout the value iteration loop.

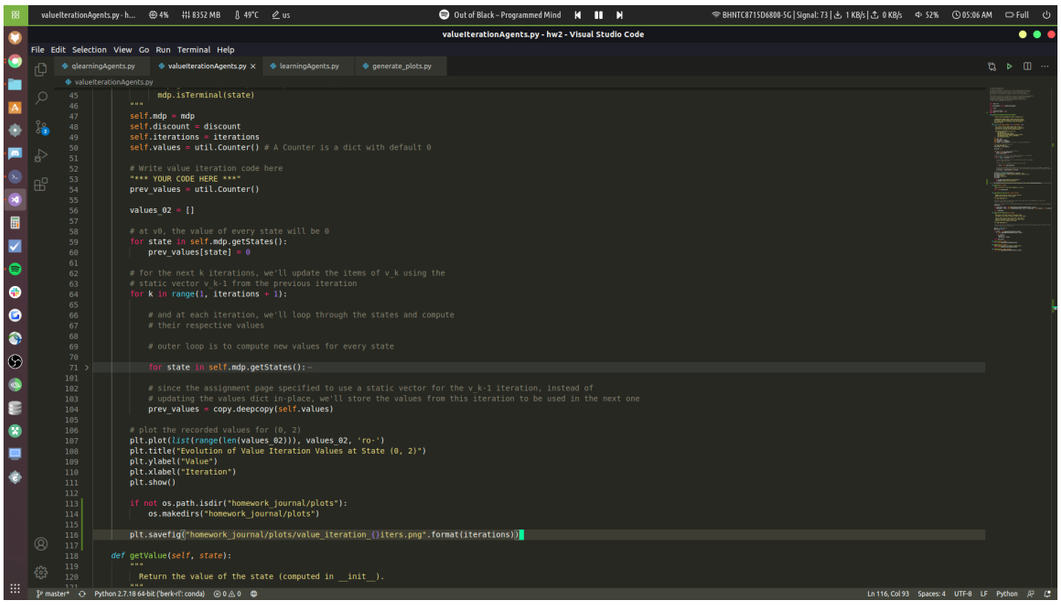


Figure 30: Code for generating my value iteration plots.

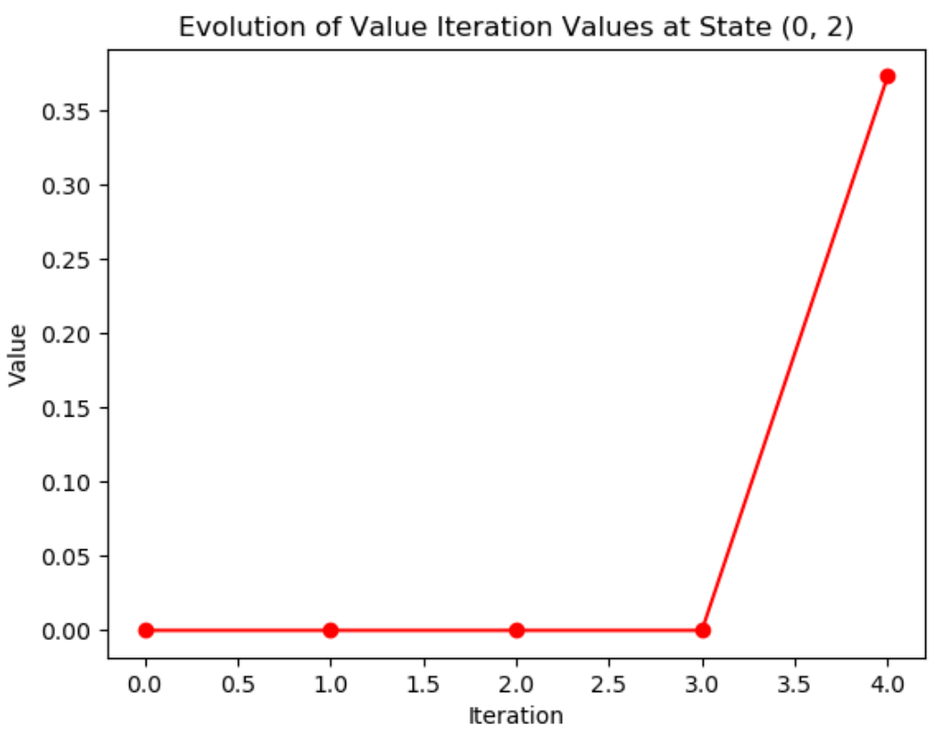


Figure 31: Value iteration plot for state (0, 2) after 5 iterations.

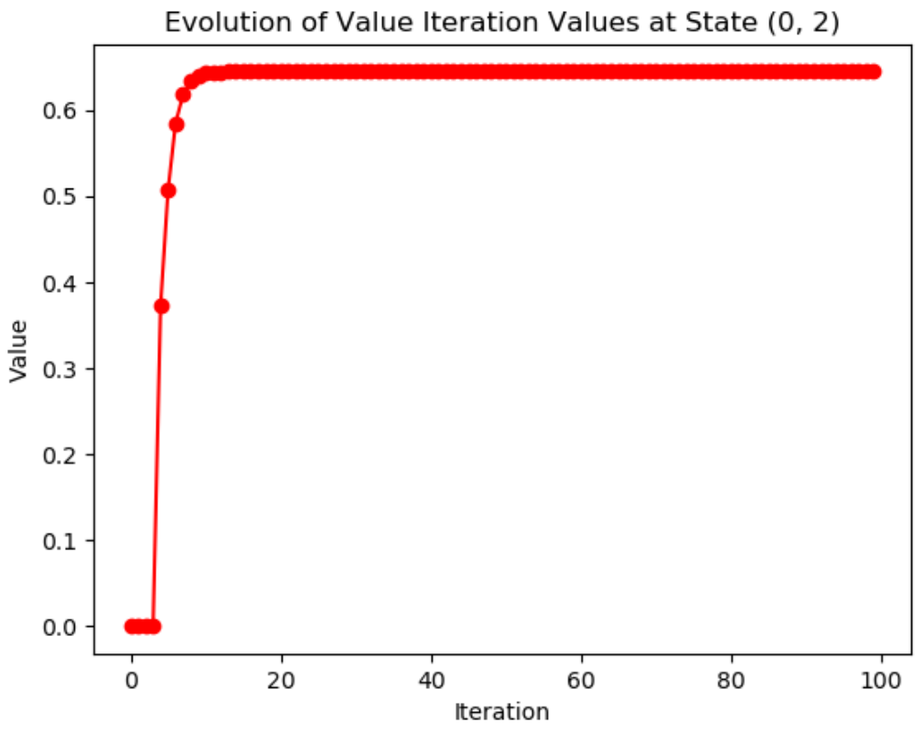


Figure 32: Value iteration plot for state (0, 2) after 100 iterations.

5.2 Q-Learning Plots

These plots were a bit more difficult to generate, since we only really have access to the `update()` method, giving us no information about the overall training scheme or ability to persistently store Q-values. So to log values at each call to `update()`, I simply save them to an external file and generate the plots later. To generate the plots yourself, you can simply uncomment my code in the `__init__()` and `update()` methods of the Q-learning agent, run the GUI demo, then run `generate_plots.py` in `homework_journal/plots`. Be sure to supply command line flags for the number of episodes and path to the CSV file containing the logged Q-values (i.e. `--episodes 100` and `--path q_vals_100eps.csv`).

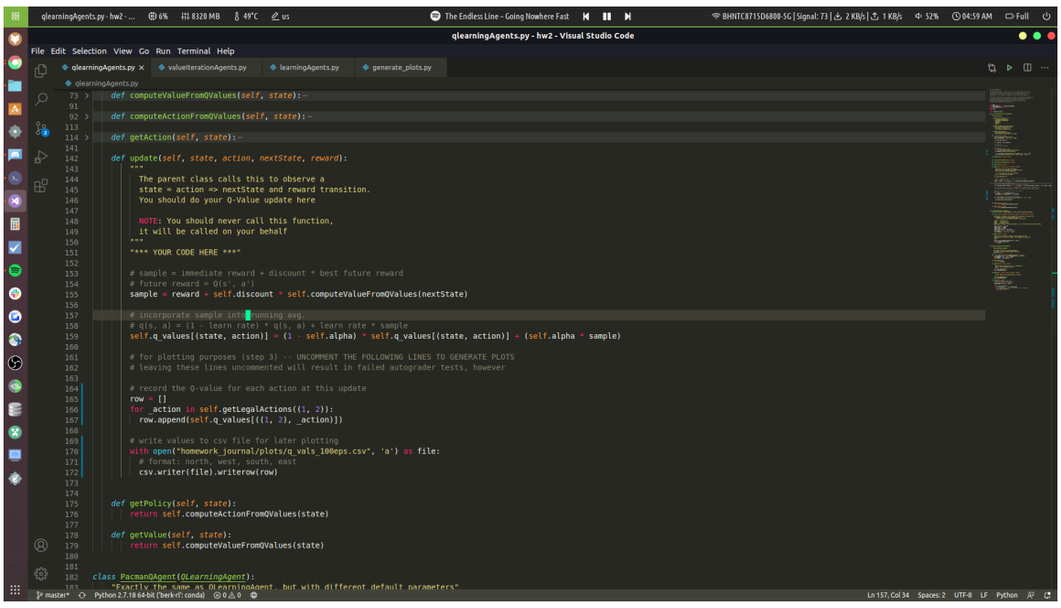


Figure 33: Code I used in `qlearningAgent.py` to save Q-values for later plot generation. For the code that generates the plots, see `homework_journal/plots/generate_plots.py`.

Finally, here are my plots for the state of (1, 2) in Q-learning with 5 episodes of training time and 100 episodes of training time. Note that, since I'm logging updates by the number of calls to `update()`, and not by episodes, the number of "iterations" may vary across multiple runs.

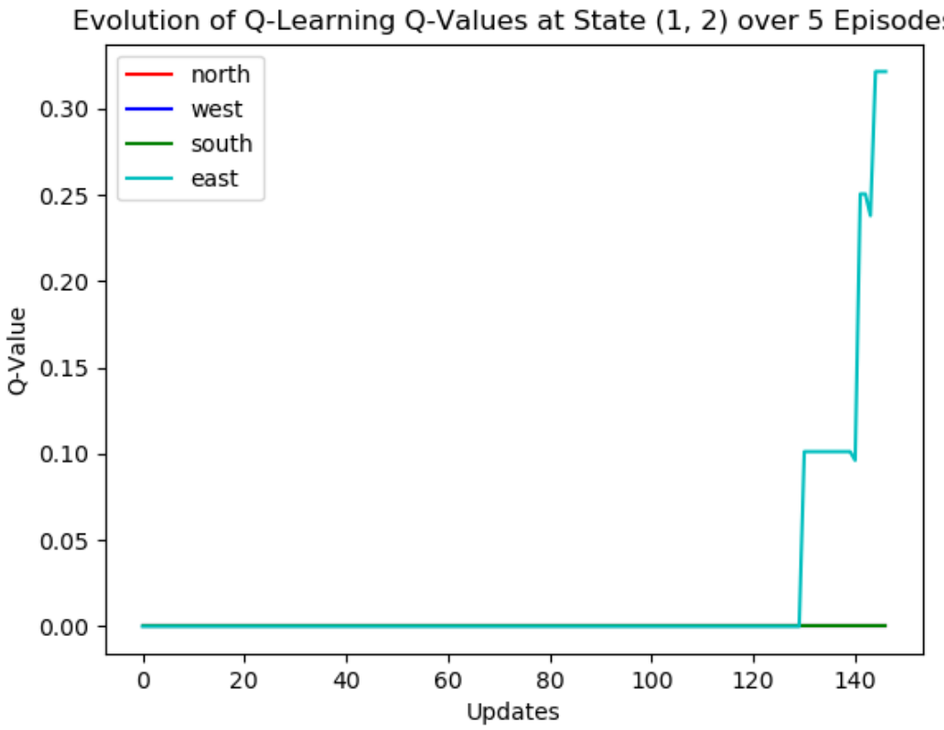


Figure 34: Q-values in the state (1, 2) over 5 episodes of training. Note that the other actions other than east are all very low (or zero), because the agent hasn't had enough time to explore these other options, and by this point has only just barely begun to reach the goal state.

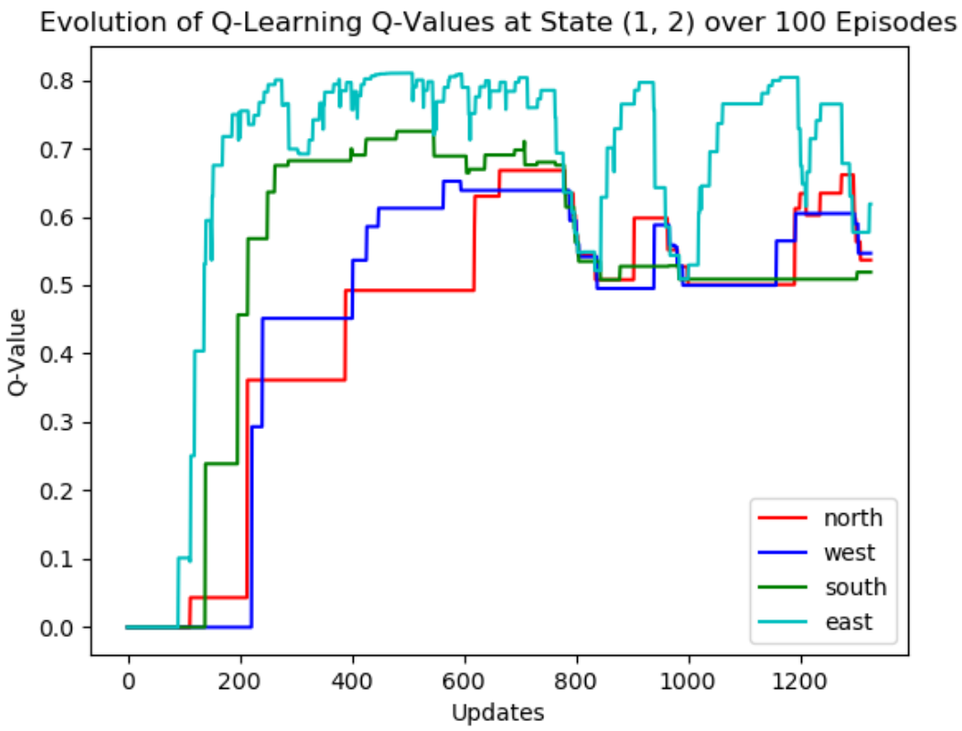


Figure 35: Q-values in the state (1, 2) over 100 episodes of training. This time around, we see more balanced distributions of performance. However, east still has higher "peaks" and ends the highest after training since it is the most direct way to the goal state. Also important to note is that, in the early-middle stages of training, these actions seem to have very large values (even suboptimal actions), but these are later culled down as the agent gathers more information from exploring.