Question 1

Introduction:

The problem statement asked for the implementation of a value iteration agent which runs a specified number of iterations to assign values to each state. It should update all the states in each iteration and at the end of the iteration, we have a value for each state.

Analysis & Explanation:

The function runs in the initialization of the class - a sort of offline planning - so that when agent initializes, each state already has a value assigned to it and the agent can follow the action that gives it the maximum value.

In each iteration, states are iterated in the order of which they are received by the function getStates(). For each state, all possible actions are iterated to get a Q-value using the equation

$$Q(s, a) = \Sigma\, P(s' \mid s, a) * [R(s, a, s') + \gamma * \max(Q(s'))]$$

The discount, probability and reward are obtained using function calls that returns previously stored values. The max q-value is taken from the array which stores value of each state and is 0 initially.

The maximum q value obtained at each state is then taken as the value of the state. Later, when an agent is at a state, it can look up the action to take by searching all the available actions and match which action produces the max q value stored for that state. In this way, policy is determined for each state.

Interesting Findings:

This existing algorithm could be improved by stopping if the value of a cell has not changed by a certain margin, which help to reduce running unnecessary iterations. Introducing a tolerance margin of some sort might speed up the process.

The algorithm does not run for terminal states, and those are assigned a value of 0. This is because when running iterations for each possible action of a state, terminal states has no available actions or only one action - exit. The value thus ends as being assigned zero.

Behavior of code for different hyper parameters:

All the parameters were evaluated based on the values assigned to the states and the average score returned.

One of the hyperparameters was the number of iterations. This was experimented with using terminal command. When changing the number of iterations to values of 10, 50 and 100, the effect was difficult to observe because running the code multiple times causes it to give different values from the range of 0.47 to 0.55. The trend becomes difficult to establish due to the fact that sometimes more iterations give more value and sometimes lower iterations give more value. It is thus clear that while iterations may have some level of effect, such as more iterations giving better q-values, the effect remains insignificant to some degree as long as the number of iterations is higher than 50. Under value of 50, the average score decreases. Of course, this is on the grid provided by gridworld.py and may become a significant factor if used on different layouts.

Another hyperparameter is the discount value used to evaluate the q values of each state. Increasing the discount value increased the values of each state, and the average score obtained was higher due to the fact that the q values were higher. Similarly, when the discount value was decreased to 0.2, the average score obtained was also lower because the values assigned to the state were lower.

Question 2

Introduction:

The problem statement asked for assigning appropriate values of discount and noise that would allow an agent to cross a bridge safely, while avoiding the fire on either side.

Analysis & Explanation:

In this problem, the optimal policy is already defined. The road to the goal is very risky since any mistake that causes agent to take any move other than the optimal move will cause it to fall into fire.

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V_k(s')\right]$$

Discount value was thus given 0.9, since a higher value will give a higher importance to future state values and will assign higher q values to the actions that leads the agent to the states with higher q values. This essentially means the agent is less likely to take risks and will always choose the states giving the highest q value, which leads to terminal states with higher end rewards. The noise is assigned value of 0, because the noise would introduce uncertainty, which we cannot afford since even a small noise value might cause the agent to end up in an unwanted state after taking the optimal action.

Behavior of code for different hyper parameters:

The code was tested using different hyperparameters using this command:

```
python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2
```

When the value of discount is set at 0, the reward achieved is 0 because the calculated q values are assigned a value of 0. Thus it points at the default policy i.e. up. When the discount value is increased upto the value of 0.5, the policy changes to point to the start state which has a reward of 1.0. This is because lower discount assigns lower importance to future rewards and more importance to gaining rewards sooner. As a result it ends up moving towards the start state again since that is the closest and highest reward. Any value of 0.6 and above causes it to move across the bridge to the desired end state with the average reward increases with the value of discount.

A noise value of 0 paired with discount value of 0.9 provided the desired value. Any noise value above that causes the policy to point back to the start state. The value of states are negative with increasing noise values causes the state q values to decrease. This is because during 100 iterations, the values assigned keep decreasing due to the negative rewards achieved each time as the agent keeps falling into the -10 grids due to noise.

It was observed that a value of above 5 iterations provide the desired policy and any value below that causes the agent to go back to the start state. Other combinations were tested and it was found the boundary values that produces the desired or correct policy are 5 iterations, noise value of 0 and discount value of 0.6.

Question 3

Introduction:

The problem statement asked for assigning appropriate values of living reward, discount and noise that would allow an agent to get to the specified goal (either one of the two) using a specified path (either the safe long path or short riskier path).

Analysis & Explanation:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V_k(s')\right]$$

The living reward causes the agent to place more importance on taking the longer road to the goal. This is because by taking the long road it will be able to collect more reward at the end since each state gives a reward, essentially rewarding the agent for exploration.

On other hand, higher discount would place more importance on reaching goals that are further away with higher value assigned to it. The future rewards are multiplied by this value thus the discount represents the amount of importance attached to the values of the states that can be reached in the future.

Finally, noise represents the uncertainty i.e. there is a probability that even after taking the optimal action, the agent may end up in a different state than the one specified in the policy.

With these factors in mind, the following cases were solved.

a. Prefer the close exit (+1), risking the cliff (-10)

This asks the agent to take a risky path to a more closer goal. With the knowledge of the influence of these factors, the living reward was given a value of 0 so agent is not encouraged to keep on exploring. A discount value of 0.2 was given to encourage agents to reach the goal but not a higher value so agent does not move towards the higher valued goal (+10). Since the path is risky, a noise value could cause agent to fall into the negatively valued states so the noise value is assigned 0. This ensures the agent ends up where the action should take it to.

b. Prefer the close exit (+1), avoiding the cliff (-1)

Since we still want to prefer the closer goal, the discount value is kept unchanged to 0.2. The living reward is increased to 0.5 to encourage the agent to take a longer path. However a noise of 0.1 is also introduced so that agent ends up moving towards the closer goal instead of moving towards the further exit because of living reward.

c. Prefer the distant exit (+10), risking the cliff (-10)

Previously, the lower values of discount made the agent prioritize immediate rewards over rewards in states further away. Thus value of discount is increased to 0.7 which is a balance between the agent wanting to reach the goal state further away with a higher value but not so high that it is not willing to take the riskier path. The noise value is kept at 0 but a value of 0.1 also works to make sure the agent does not end up in the negatively valued states by accident. Again, the living reward is kept low at a value of 0.2 so that agent does not explore.

d. Prefer the distant exit (+10), avoiding the cliff (-10)

The same values for the previous solution was used but living reward was increased so agent would explore the longer path to reach the goal. As before, the high discount causes agent to prioritize the higher value goal at a distance and low noise ensures agent reaches goal with little thrashing around. If noise is increased, the agent thrashes around but eventually reaches the +10 through the long way.

e. Avoid both exits and the cliff (so an episode should never terminate)

The living reward is increased to 1 while discount value is assigned at 0 so that all the states are assigned value 0.9 equally. The discount value 0 also ensures that agent does not place any importance in future goals which is why it does not move towards the closer or further away goal state. The noise value does not have much importance here as lower noise value means it stays in one place and higher noise value means it thrashes around but in the end, the episode does not terminate.

Challenges:

Since there were 3 factors influencing each other and the policy, it was difficult to grasp the relationship between them well enough to understand why a particular combination of values worked and why some did not. Achieving the answer was easy but understanding the

reason behind it was a bit tricky. However, once I comprehended the influence of each factor on the q values calculated using the formula, it was easy to understand which combination of values might produce the result the problem statement asked for.

Question 4

Introduction:

The problem statement asked for modification of the solution to the first problem by updating one state per iteration as opposed to updating all states in each iteration. The question instructs to ignore terminal state and update state in the order provided by getStates() function.

Analysis & Explanation:

Iteration is done as usual but before iterating the a current state index is initialized at 0 and number of states is determined using the length of the state array returned by the getStates() function. Then in each iteration, using the current state index, the state is accessed and updated accordingly. Each time the current index is incremented by 1 and mod by the number of states to ensure cyclic value iteration. This is repeated until the number of iterations complete.

Challenges:

The cyclic iteration was a bit challenging to think of but mod operation solved this issue. Another alternative way I could have done the cyclic iteration would have been to check if the current state index was equal to the number of states, which would mean one rotation of all the states had been completed. If the check was true, then the state index could have been set back to 0 to start from the beginning.

Question 5

Introduction:

The problem statement asked for the modification of the first solution to implement a prioritized sweeping algorithm.

Analysis & Explanation:

This algorithm makes the process of state update more efficient. Instead of updating all states, it only updates states when the difference between new and previous q value is large enough to cause a change in the policy.

A priority queue is taken to keep track of the states that need to be updated. Priority is set by iterating over all the states once and based on the negative of the difference between the current and newly computed q value (since the priority queue is min-priority queue by default), which is the maximum of the q values calculated for each action. At the start the current value is set to zero. This ensures that states which have the largest difference between current and old value is prioritized for state update since it might cause a change in the policy.

Next a number of iterations is run, where number of iterations is predefined by the variable defined in the class. The state at the top of the priority queue is selected for update and its q value is calculated and the maximum q value for that state is updated. Since, according to the Bellman equation, the predecessor state values might change due to the change in this value, the state's predecessor is iterated one by one. For each predecessor, its new q value is calculated and the difference is calculated between the new and previous q value. If it above a predefined tolerance level theta, the state value is updated. Only one level of predecessor is updated because Bellman equation says the value of a state is only dependent on its predecessor.

Challenges:

It was difficult to understand the paper and implement the algorithm stated. However, with a bit of patience and some time, it was possible to break it down into smaller problems that could be implemented to give the desired solution to this problem.