CMPSCI 687 - Extra Credit Assignment (Optional) - Fall 2023

Due **December 05, 2023**, 11:55pm Eastern Time

1 Instructions

This assignment consists only of a programming portion. While you may discuss problems with your peers (e.g., to discuss high-level approaches), you must answer the questions on your own. In your submission, do explicitly list all students with whom you discussed this assignment. Submissions must be typed (handwritten and scanned submissions will not be accepted). You must use LaTeX. The assignment should be submitted on Gradescope as a PDF with marked answers via the Gradescope interface. The source code should be submitted via the Gradescope programming assignment as a .zip file. Include with your source code instructions for how to run your code. You **must** use Python 3 for your homework code. You may not use any reinforcement learning or machine learning-specific libraries in your code, e.g., TensorFlow, PyTorch, or scikit-learn. You *may* use libraries like numpy and matplotlib, though. The automated system will not accept assignments after 11:55pm on December 05. The tex file for this homework can be found here.

Before starting this homework, please review this course's policies on plagiarism by reading Section 10 of the syllabus.

Programming (65 Points Total)

In this assignment, you will implement three algorithms based on Temporal Difference (TD): TD-Learning for policy evaluation; SARSA; and Q-Learning. You will deploy all three algorithms on the 687-Gridworld. **Notice that you may not use existing RL code for this problem—you must implement the learning algorithms and the environments entirely on your own and from scratch.** The complete definition of the 687-GridWorld domain can be found in Homework 3.

General instructions:

- You are free to initialize the q-function in any way that you want. More details about this later.
- Whenever displaying values (e.g., the value of a state), use 4 decimal places; e.g, $v(s) = 9.4312$.
- Whenever showing the value function and policy learned by your algorithm, present them in a format resembling that depicted in Figure 1.

Value Function

4.0187	4.5548	5.1575	5.8336	6.4553
4.3716	5.0324	5.8013	6.6473	7.3907
3.8672	4.3900	0.0000	7.5769	8.4637
3.4182	3.8319	0.0000	8.5738	9.6946
2.9977	2.9309	6.0733	9.6946	0.0000

Policy

→	→	→	↓	↓
→	→	→	↓	↓
↑	↑		↓	↓
↑	↑		↓	↓
↑	↑	→	→	G

Figure 1: Optimal value function and optimal policy for the 687-GridWorld domain.

1. TD-Learning on the 687-Gridworld [14 Points, total]

First, implement the TD-Learning algorithm for policy evaluation and use it to construct an estimate, $\hat{v}$, of the value function of the optimal policy, π^*, for the 687-Gridworld. **Remember that both the optimal value function and optimal policy for the 687-Gridworld domain were identified in a previous homework by using the Value Iteration algorithm. For reference, both of them are shown in Figure 1.** When sampling trajectories, set d_0 to be a uniform distribution over $\mathcal{S}$ to ensure that you are collecting returns from all states. Do not let the agent be initialized in an Obstacle. Notice that using this alternative definition of d_0 (instead of the original one) is important because we want to generate tr. jectories from all states of the MDP, not just states that are visited under the optimal deterministic policy shown in Figure 1.

You should run your algori. hm until the value function estimate does not change significantly between successive iterations; i.e., whenever the Max-Norm between $\hat{v}_{t-1}$ and $\hat{v}_t$ is at most δ, for some value of δ that you should set empirically. For each run of your algorithm, count how many episodes were necessary for it to converge to an estimate of v^{π^*}. **You should run your algorithm (as described above) 50 times and report the value of α that was used. You should pick a value of α that causes the algorithm to converge to a solution that is close to v^{π^*}, while not requiring an excessively large number of episodes.** Then:

(Question 1a. 7 Points) Report the average value function estimated by the algorithm—i.e., the average of all 50 value functions learned by the algorithm at the end of its execution. Report, also, the Max-Norm between this average value function and the optimal value function for the 687-Gridworld domain.

Design choices: Max norm δ considered was 0.001. The γ value is 0.9. The step size used was $\frac{0.75}{\sqrt{i}+1}$, where i is the number of episodes completed so far. The square root factor ensures a gradual decay of step size, and makes sure that step size doesn't get too small with number of episodes. Grid search was performed on step sizes and δ values, with δ of 0.001 and gradual decay of step size found to work the best.

The average value function estimated by the algorithm is,

$$\begin{bmatrix} 3.9880 & 4.5404 & 5.1532 & 5.8399 & 6.4214 \\ 4.3655 & 5.0367 & 5.8065 & 6.6583 & 7.3985 \\ 3.8472 & 4.3888 & 0. & 7.5901 & 8.4877 \\ 3.3872 & 3.8134 & 0. & 8.6017 & 9.7215 \\ 2.9867 & 2.9204 & 6.1085 & 9.7541 & 0. \end{bmatrix}$$

The max norm is 0.0507

(Question 1b. 7 Points) Report the average and the standard deviation of the number of episodes needed for the algorithm to converge.

The average number of episodes was 29543. The standard deviation observed of the number of episodes taken was 5256.

2. SARSA on the 687-Gridworld [18 Points, total]

Implement the SARSA algorithm and use it to construct $\hat{q}$, an estimate of the optimal **action-value function**, q^{π^*}, of the 687-Gridworld domain; and to construct an estimate, $\hat{\pi}$, of its optimal policy, π^*. You will have to make four main design decisions: *(i)* which value of α to use; *(ii)* how to initialize the q-function; you may, e.g., initialize it with zeros, with random values, or optimistically. Remember that $q(s_\infty, \cdot) = 0$ by definition; *(iii)* how to explore; you may use different exploration strategies, such as ϵ-greedy exploration or softmax action selection; and *(iv)* how to control the exploration rate over time; you might choose to keep the exploration parameter (e.g., ϵ) fixed over time, or have it decay as a

function of the number of episodes. **Report the design decisions *(i, ii, iii, and iv)* you made and discuss what was the reasoning behind your choices.** Then:

Design decisions:

(a) α **value used:** The step size used was a constant step size of 0.1. The observation made was that much decaying step sizes over time works, only as long as the exploration rate doesn't dominate the step size (as the step size is what governs the update rule). Another step size that worked well in practice was $\frac{0.5}{\sqrt{i}+1}$ where i is the total number of episodes completed by the agent so far in a given run. Too high a step size may lead to high variability and oscillations, and we may even end up overshooting on the q-values. Higher step sizes tend to prioritize on more recent outcomes/rewards, while lower step sizes means the agent keeps into account a longer context of rewards obtained over time and in turn gradual and slow updates.

(b) **q function initialization**: In this case, initializing to a uniform distribution between 0 and 1 worked best in practice. Zero initialization also worked equally well. Too optimistic an initialization (for eg: q-values as high as 100 or even 1000) may take much longer for the agent to converge, if a constant step size of 0.1 is used, and so such initializations must be avoided. Too high a q-value can lead to more exploration, and in turn take longer for the agent to converge. Similarly, biased initialization of q-values can cause the agent to get stuck in a local minima.

(c) ϵ **value used:** The starting value of epsilon used was 0.8. This encourages early exploration in an ϵ-greedy action selection setting. Too high an ϵ value means the agent takes random actions most of the time, and never really learns. Even on starting with a higher ϵ value, it is a good practice to decay the ϵ to facilitate convergence.

(d) ϵ **decay:** The starting value of epsilon used was 0.8. It was decayed by a factor of 2 every 1000 episodes in a given run. This encourages early exploration, and as the agent quickly picks up the optimal Q-values, exploration rate is decreased, and exploitation increases, as ϵ decays exponentially. Decaying ϵ is crucial, so as to have more exploitation and enable the agent to complete a given episode in as little actions as possible, once it has figured out the optimal policy.

(Question 2a. 7 Points) Construct a learning curve where you show, on the x axis, the total number of actions taken by the agent, from the beginning of the training process (i.e., the total number of steps taken across all episodes up to that moment in time). On the y axis, you should show the number of episodes completed up to that point. If learning is successful, this graph should have an increasing slope, indicating that as the agent takes more and more actions (and thus executes more learning updates), it requires fewer actions/timesteps to complete each episode. Your graph should, ideally, look like the figure shown in Example 6.5 of the RL book (2nd edition). To construct this graph, run your algorithm 20 times and show the average curve across those runs.

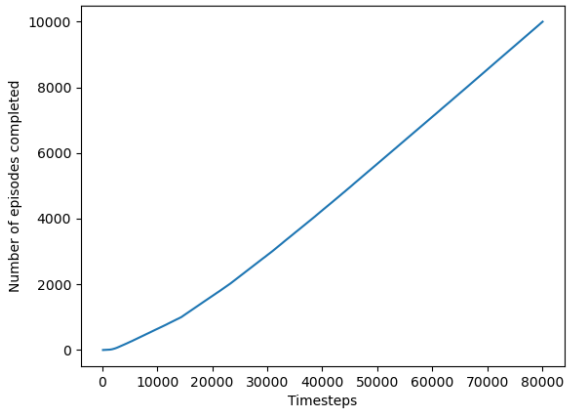

Figure 2: Number of episodes completed vs number of actions taken

As observed, there is a en early spike at approximately t=50, after which there is a steady increase in the number of episodes completed over time.

(Question 2b. 8 Points) Construct another learning curve: one where you show the number of episodes on the x axis, and, on the y axis, the mean squared error between your current estimate of the value function, $\hat{v}$, and the true optimal value function, v^{π^*}. The mean squared error between two value functions, v_1 and v_2, can be computed as $\frac{1}{|\mathcal{S}|}\sum_s (v_1(s) - v_2(s))^2$. Remember that SARSA estimates a q-function ($\hat{q}$), however, not a value function ($\hat{v}$). To compute $\hat{v}$, remember that $\hat{v}(s) = \sum_a \pi(s,a)\hat{q}(s,a)$. Remember, also, that the SARSA policy π, at each given moment, depends on your exploration strategy. Let us say, for example, that you are using ϵ-greedy exploration. Then, in the general case when more than one action may be optimal with respect to $\hat{q}(s,a)$, the ϵ-greedy policy would be as follows:

$$\pi(s,a) = \begin{cases} \frac{1-\epsilon}{|\mathcal{A}^*|} + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a \in \mathcal{A}^* \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise,} \end{cases} \tag{1}$$

where $\mathcal{A}^* = \arg\max_{a \in \mathcal{A}} \hat{q}(s,a)$. To construct the learning curve, run your algorithm 20 times and show the average curve across those runs; i.e., the average mean squared error, computed over 20 runs, as a function of the number of episodes.

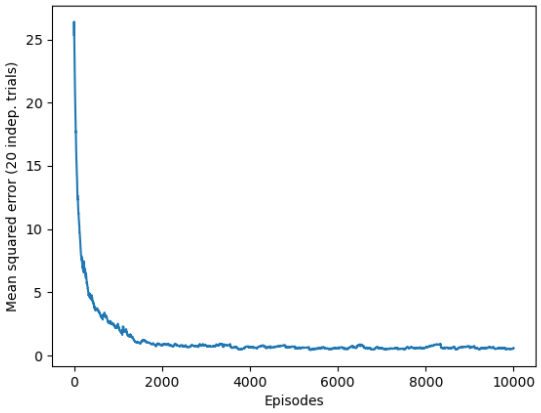

Figure 3: Mean squared error across 10000 episodes

(Question 2c. 3 Points) Show the *greedy policy* with respect to the q-values learned by SARSA.

Figure 4: Policy learnt by the algorithm

3. Q-Learning on the 687-Gridworld [18 Points, total]

Answer the same questions as above (*2a, 2b, and 2c*), but now using the Q-Learning algorithm; i.e., you will be using Q-Learning to solve the 687-Gridworld domain. When computing $\hat{v}$, for question *3c*, you should compute the value function of the greedy policy: $\hat{v}(s) = \max_a \hat{q}(s, a)$. The amount of points associated with questions *3a, 3b, and 3c*, will be the same as those associated with the corresponding items of Question 2.

Design decisions:

(a) α **value used:** The step size used was a constant step size of 0.1. The observation made was that much decaying step sizes over time works, only as long as the exploration rate doesn't dominate the step size (as the step size is what governs the update rule). Another step size that worked well in practice was $\frac{0.5}{\sqrt{i}+1}$ where i is the total number of episodes completed by the agent so far in a given run. Too high a step size may lead to high variability and oscillations, and we may even end up overshooting on the q-values. Higher step sizes tend to prioritize on more recent outcomes/rewards, while lower step sizes means the agent keeps into account a longer context of rewards obtained over time and in turn gradual and slow updates.

(b) **q function initialization**: In this case, initializing to a uniform distribution between 0 and 1 worked best in practice. Zero initialization also worked equally well. Too optimistic an initialization (for eg: q-values as high as 100 or even 1000) may take much longer for the agent to converge, if a constant step size of 0.1 is used, and so such initializations must be avoided. Too high a q-value can lead to more exploration, and in turn take longer for the agent to converge. Similarly, biased initialization of q-values can cause the agent to get stuck in a local minima.

(c) ϵ **value used:** The starting value of epsilon used was 0.8. This encourages early exploration in an ϵ-greedy action selection setting. Too high an ϵ value means the agent takes random actions most of the time, and never really learns. Even on starting with a higher ϵ value, it is a good practice to decay the ϵ to facilitate convergence.

(d) ϵ **decay:** The starting value of epsilon used was 0.8. It was decayed by a factor of 2 every 1000 episodes in a given run. This encourages early exploration, and as the agent quickly picks up the optimal Q-values, exploration rate is decreased, and exploitation increases, as ϵ decays exponentially. Decaying ϵ is crucial, so as to have more exploitation and enable the agent to complete a given episode in as little actions as possible, once it has figured out the optimal policy.

- **Question 3a** The plots are outlined as below, the first plot shows the log of total number of actions taken against number of episodes. The seocnd shows the total number of actions taken against number of episodes elapsed upto that point.

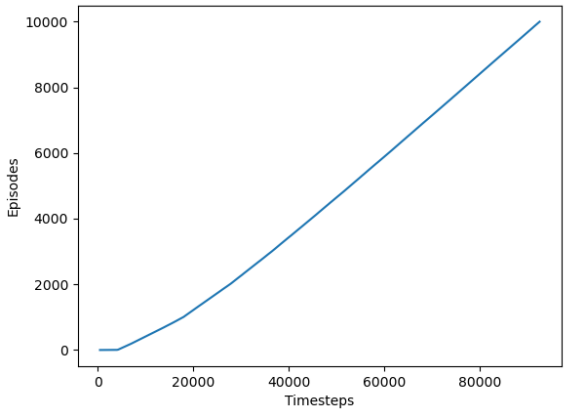

Figure 5: Enter Caption

- **Question 3b** The mean squared error plot is as follows,

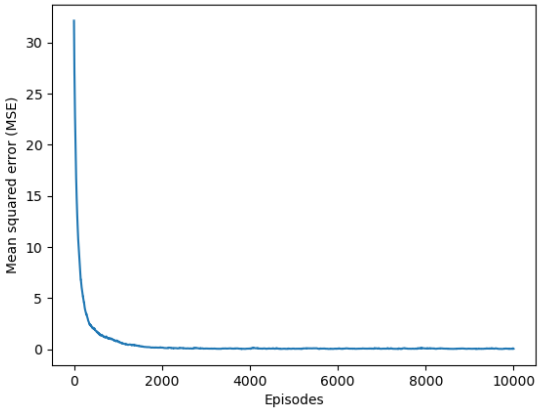

Figure 6: Mean squared error across 10000 episodes

- **Question 3c** The greedy policy learnt by the algorithm is as follows,

Figure 7: Policy learnt by the algorithm

4. Creating Your Own Gridworld [15 Points, total]

In this part of the assignment, you will investigate different properties of RL algorithms using a pilot program developed by one of the TAs. This program allows you to easily and interactively create you own gridworld MDPs. First, clone this repository and follow the installation instructions. All instructions for using the program are in the repository's README.

(Question 4a. 6 Points) You will first investigate how different definitions of the initial state distribution, d_0, may affect the difficulty of solving reinforcement learning problems. Start by launching the program and use it to define the components of your MDP: $\mathcal{S}$, p, d_0, $R(s')$, and γ. Notice that you can solve this MDP (i.e., find an optimal policy) via the Value Iteration algorithm by clicking the "Solve!" option. Once you have done so, the program will display the number of iterations, n, required for Value Iteration to converge. Next, click "Show Value Function" to display the optimal value function, v^*, identified by Value Iteration, as well as the expected return of the corresponding optimal policy; that is, J^*.

You will now create your own MDP and investigate two initial state distributions, d_0 and d'_0, which you are free to specify. Let n be the number of iterations required for Value Iteration to converge. Your goal is to identify/specify d_0 and d'_0 such that:

- When using d_0, it takes SARSA *fewer* than n episodes to achieve an average return that corresponds to 80% of the maximum return possible, that is $0.8\,J^*_{d_0}$, where $J^*_{d_0}$ is the expected return of the optimal policy when using the initial state distribution d_0.
- When using d'_0, it takes SARSA *more* than $10n$ episodes to achieve an average return that corresponds to 80% of the maximum return possible, that is $0.8\,J^*_{d'_0}$, where $J^*_{d'_0}$ is the expected return of the optimal policy when using the initial state distribution d'_0.

When solving this question, please use the program's default hyperparameters for SARSA. Furthermore, notice that the *average return* mentioned above can be easily determined by visually inspecting the exponential moving average (EMA) line plotted along with the learning curves presented by the program.

Given the MDP you created and the two initial state distributions you specified:

- Present a screenshot of your MDP;
- Present the learning curves of SARSA for each of the initial state distributions, d_0 and d'_0;
- Describe in English what the distributions you designed, d_0 and d'_0, are modeling and why you chose them;
- Report the value of n. Also report $J^*_{d_0}$ (the expected return of the optimal policy for the MDP with initial state distribution d_0) and $J^*_{d'_0}$ (the expected return of the optimal policy for the MDP with initial state distribution d'_0).

ANSWER:

- The MDP designed is identical to the 687-Gridworld MDP. Possible states are from (0,0) to (4,4), with obstacles at (3,2) and (2,2), a water state at (4,0) and goal state at (4,4). The water state has a reward of -10, the goal state has a reward of +10, and all other states have reward of 0.
- Screenshots of the MDP are presented below,

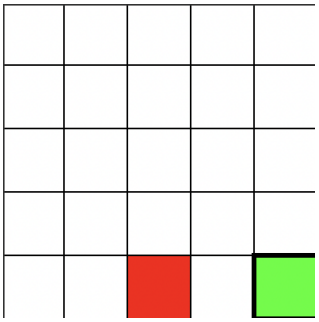

Figure 8: Rewards defined

- **Answer - Part A:**
  - It took 19 iterations for value iteration to converge. n=19, in this case.
  - The expected return of the optimal policy obtained in value iteration is 9.6916.
  - The expected return estimate of optimal policy, obtained with the SARSA algorithm is 8.29 at the end of 1000 episodes.
  - **Observation**: The SARSA algorithm gets to higher than 80 percent of expected return from value iteration in 12 iterations, with an estimate of 7.96.

– The initial state distribution however is designed such that the agent only starts at (3,4), a state which is very close to the goal state. So the probability of starting in any other state apart from (3,4) is 0. The initial state distribution is shown below,

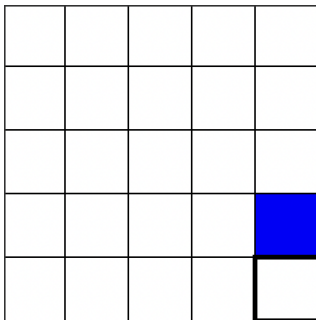

Figure 9: Possible start states

– The SARSA curve is presented below,

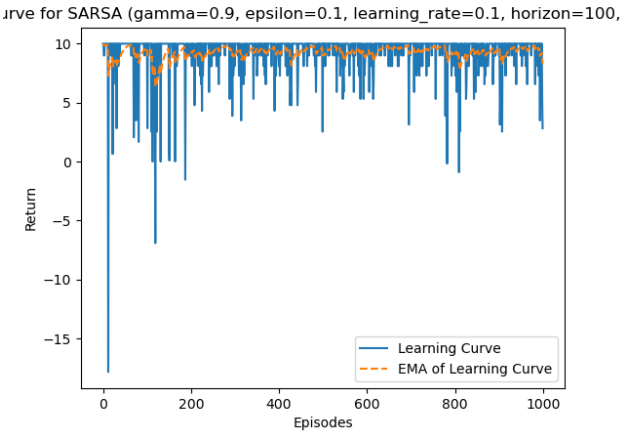

Figure 10: SARSA learning curve

– The policy learnt is shown below,

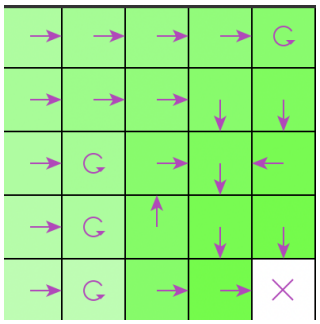

Figure 11: Optimal policy learnt by SARSA

As expected, the policy learnt is erroneous, because (3,4) is the only possible initial state that is being generated as part of the trajectories.

- **Reasoning:** When setting the initial state distribution such that (3,4) is the only possible initial state, it is very likely that the agent almost never encounters the water state, and overestimates the quantity $J^{\star}$. This is observed in the SARSA learning curve, where the agent over-estimates the value to be above 9.69 (which is the value function estimate obtained for the optimal policy, under value iteration) at most episodes. Note that value iteration algorithm includes transition probabilities, which SARSA does not. SARSA constructs its estimates by several interactions with the environment, and it is possible to construct a maximum likelihood estimate of transition prob. matrix with interactions obtained through SARSA. In this case, however, with start state distribution skewed and (3,4) being the only start state possible, we get over-estimates of the value function, and hence it achieves 80 percent of return obtained through value iteration quite soon.

- **Answer - Part B:**
  - The initial state distribution is as mentioned below,

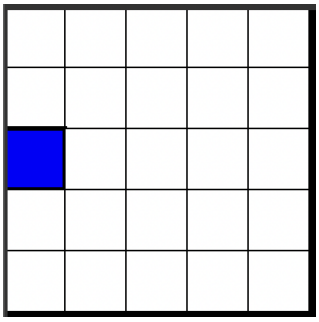

Figure 12: Initial State distribution

  - It took 19 iterations for the value iteration to converge. n =19 in this case.
  - The expected return of the optimal policy obtained through value iteration is 3.9254.
  - The expected return obtained through SARSA algorithm for the optimal policy is 2.33 at the end of 1000 episodes.
  - At the end of $10n$ episodes, the algorithm manages to yield a return of 2.93, which is less than 80 percent of 3.9254.
  - The SARSA curve is presented below,

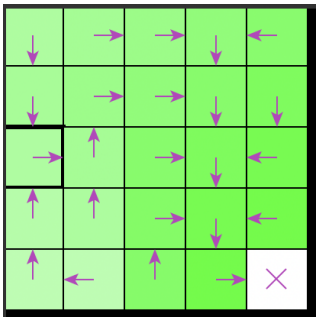

Figure 13: Policy learnt for the initial state distribution

  - **Reasoning:** When setting the initial state distribution d_0 such that (2,0) is the only possible initial state, the agent has to figure out a longer route to get to the goal state. Also with the water state being closer to the agent, than the goal state from (2,0) the agent has to also find a way to avoid going very close to the water, which is the expected behavior with the SARSA algorithm. The factor of γ is also very important to keep in mind, as a high factor leads to

severe diminishing returns particularly when the goal state has to be reached through a longer route. Given these factors, the agent takes longer to converge to the optimal value function estimate. Below is a screenshot of the optimal policy learnt by the algorithm,

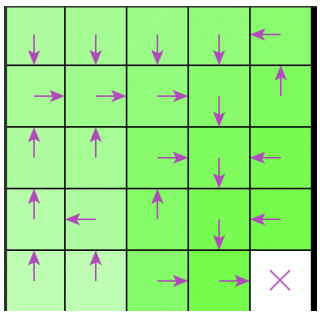

Figure 14: Optimal policy learnt by SARSA

(Question 4b. 6 Points) You will now investigate how the values used to initialize the q-function being learned by Q-Learning impact its learning speed. To do so, either use the same MDP as in the previous question (and select only one of the initial state distributions you designed) or create a new MDP. Your goal is to identify two ways of initializing the q-function being learned by Q-Learning: q_0 and q_0'. You should identify these q-function initializations such that:

- When using q_0 as its initial q-function, it takes Q-learning *fewer* than $3n$ episodes to achieve an average return that corresponds to 80% of the maximum return possible, that is $0.8\,J^*$;
- When using q_0' as its initial q-function, it takes Q-learning *more* than $10n$ episodes to achieve an average return that corresponds to 80% of the maximum return possible, that is $0.8\,J^*$.

When solving this question, you can use any Q-Learning hyperparameters you like. As before, ***(i)*** present a screenshot of your MDP (unless it is the same as you defined in the previous question); ***(ii)*** present the learning curves of Q-Learning when using q_0 as its initial q-function, and when using q_0' as its initial q-function. Clearly identify which learning curve is associated with which initial q-function; and ***(iii)*** report the values of n, J^*, d_0, and p.

ANSWER:

- The MDP designed is identical to the 687-Gridworld MDP. Possible states are from (0,0) to (4,4), with obstacles at (3,2) and (2,2), a water state at (4,0) and goal state at (4,4). The water state has a reward of -10, the goal state has a reward of +10, and all other states have reward of 0. The only possible start state is (0,0).
- Screenshots of the MDP are presented below,

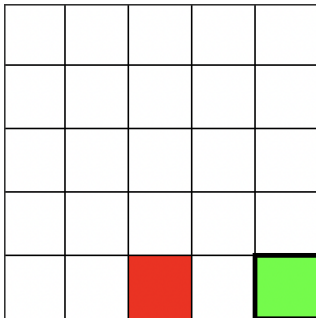

Figure 15: Rewards defined

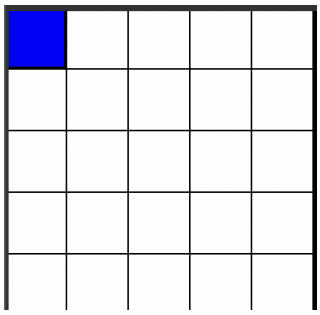

Figure 16: Possible start states

- **Answer - Part A:**
  - Value iteration took 19 iterations to converge. n =19 in this case.
  - The value function estimate obtained through Q-Learning is 3.30 at the end of 1000 episodes.
  - The initial state distribution is such that, (0,0) is the only possible initial state. So all trajectories during the q-learning process have (0,0) as their initial state.
  - We initialize all q-values to be at 0 which also happens to be the default.
  - The optimal value function estimate is obtained to be, 3.964.
  - The learning curve obtained for Q-Learning is as follows,

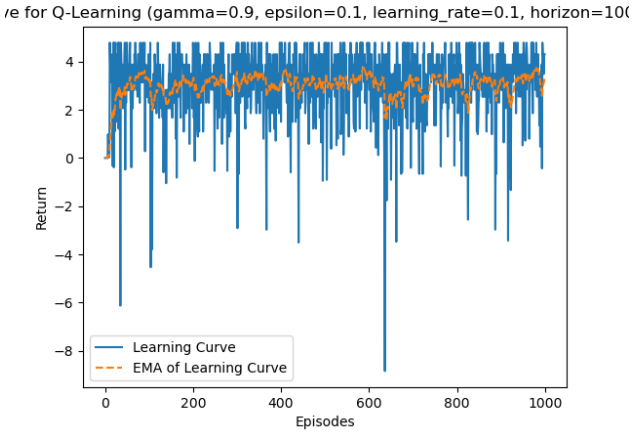

Figure 17: Q-Learning curve

  - **Reasoning:** By setting (0,0) to be the only possible initial state, we ensure all possible trajectories have (0,0) as their starting state. According to the law of large numbers, the estimate for every state converges in the limit, as we obtain more and more trajectories. Given that all trajectories start with (0,0), we are more likely to obtain convergence for this initial state. This is observed with 80 percent of the value function estimate obtained at the 52nd episode (less than 57 episodes, which is $3n$). By initializing all q-values at 0, we are starting with an initialization that is unbiased in nature (not to be confused with unbiased estimator), and this leads to faster convergence, unlike more more optimistic initial values. As is the case with Q-Learning algorithms, the agent learns a path to the gold state even while coming close to the water state. This is due to the nature of the update equation, which has a $\max_{a'} q(s', a')$ term, meaning that the update takes place in accordance with the best possible action at the next state.

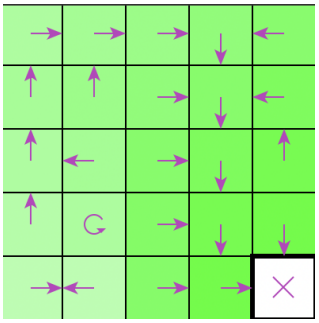

Figure 18: Q-Learning policy

- **Answer - Part B:**
  - Value iteration took 19 iterations to converge. n =19 in this case.
  - The initial state distribution is such that, (0,0) is the only possible initial state. So all trajectories during the q-learning process have (0,0) as their initial state.
  - We initialize all q-values to be at 100 which is an extremely optimistic estimate.
  - The optimal value function estimate is obtained through value iteration is 3.964.
  - The optimal value function estimate obtained through Q-Learning is, 3.46 at the end of 1000 iterations.
  - The moving average estimate of reward is close to 0, at the end of 10n episodes, this is because of the optimistic initialization, which enrourages excessive exploration.
  - The learning curve in this case is shown below,

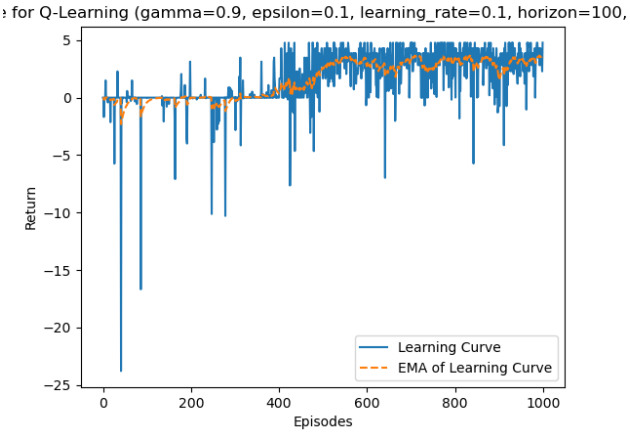

Figure 19: Q-Learning curve

  - **Reasoning:** By setting all the q-values to be 100, we set it to very optimistic initial estimates. With a fairly low step size of 0.1, the update process is slow, and the agent takes a long time to recover from its initial q-value estimates to eventually avoid the water states. The optimistic high q-value estimates promotes excessive exploration, at the cost of convergence time.
  - The policy learnt is shown below,

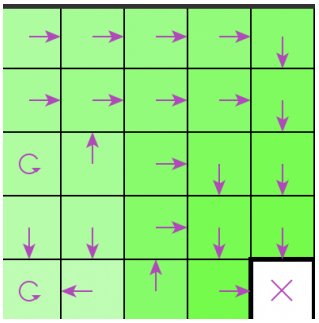

Figure 20: Policy learnt by SARSA

We observe that although the agent takes a considerably long time (around 540 episodes, i.e $\geq 10n$ episodes) it still ends up learning the optimal policy, and exhibits a behavior that is expected of q-learning, by going to the goal state, and not necessarily trying to sway too much away from the water state.

(Question 4c. 3 Points) Finally, please fill out our (short!) survey. The responses you submit are not anonymous since they will be used to grade this assignment. Importantly, however, notice that we will not associate any feedback you provide with your assignment: your feedback will only be used to improve the program designed by the TA. It will *not* impact your grade. Thank you!

Submitted feedback.