ECE59500RL HW3

Robert (Cars) Chandler — chandl71@purdue.edu

Problem 1

1.1

Using the definition of the state-action occupancy measure under a stationary, stochastic policy:

$$\nu^{\pi}_{\mu_0}(s,a) = \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(S_t = s, A_t = a | \mu_0, \pi, P)$$

The state-action occupancy measure takes a state and action pair and calculates the total (discounted) probability that the state and action at any time are that state and action. If we sum this across all actions, then we are really just finding the discounted probability that the state is that current state for all timesteps.

What we are saying below is the probability of the first state being the state in the parameter plus the sum of the probability of transitioning from some state-action combo to the state in question times the total probability across all steps of that state-action combo, across all state-action combos.

$$\sum_{a \in \mathcal{A}} \nu^{\pi}_{\mu_0}(s,a) = \mu_0(s) + \gamma \sum_{s' \in \mathcal{S}} \sum_{a' \in \mathcal{A}} P(s|s',a') \nu^{\pi}_{\mu_0}(s',a')$$

If we sum over all actions:

$$\sum_{a \in \mathcal{A}} \nu^{\pi}_{\mu_0}(s,a) = \sum_{a \in \mathcal{A}} \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(S_t = s, A_t = a | \mu_0, \pi, P)$$

Then we can drop the $A_t = a$ part since sum of the probability of a state-action pair across all possible actions will marginalize out the action on the probability measurement. From there, we can pull out the first term in the sum, marginalize the resulting probability inside the sum

across all possible states and actions of the previous timestep, rearrange our sums, and realize that we have another state-action occupancy measure defined:

$$
\begin{aligned}
\sum_{a\in\mathcal{A}} \nu^{\pi}_{\mu_0}(s,a) &= \sum_{a\in\mathcal{A}}\sum_{t=0}^{\infty}\gamma^t\mathbb{P}(S_t=s,A_t=a|\mu_0,\pi,P)\\
&= \sum_{t=0}^{\infty}\gamma^t\mathbb{P}(S_t=s|\mu_0,\pi,P)\\
&= \gamma^0\mathbb{P}(S_0=s|\mu_0,\pi,P)+\sum_{t=1}^{\infty}\gamma^t\mathbb{P}(S_t=s|\mu_0,\pi,P)\\
&= \mu_0(s)+\sum_{t=1}^{\infty}\gamma^t\mathbb{P}(S_t=s|\mu_0,\pi,P)\\
&= \mu_0(s)+\sum_{t=1}^{\infty}\gamma^t\sum_{s'\in\mathcal{S}}\sum_{a'\in\mathcal{A}}\mathbb{P}(S_t=s,S_{t-1}=s',A_{t-1}=a'|\mu_0,\pi,P)\\
&= \mu_0(s)+\sum_{t=1}^{\infty}\gamma^t\sum_{s'\in\mathcal{S}}\sum_{a'\in\mathcal{A}}\mathbb{P}(S_t=s|S_{t-1}=s',A_{t-1}=a',\mu_0,\pi,P)\cdot\mathbb{P}\left(S_{t-1}=s',A_{t-1}=a'|\mu_0,\pi,P\right)\\
&= \mu_0(s)+\sum_{t=1}^{\infty}\gamma^t\sum_{s'\in\mathcal{S}}\sum_{a'\in\mathcal{A}}P(s|s',a')\cdot\mathbb{P}\left(S_{t-1}=s',A_{t-1}=a'|\mu_0,\pi,P\right)\\
&= \mu_0(s)+\sum_{s'\in\mathcal{S}}\sum_{a'\in\mathcal{A}}P(s|s',a')\cdot\sum_{t=1}^{\infty}\gamma^t\mathbb{P}\left(S_{t-1}=s',A_{t-1}=a'|\mu_0,\pi,P\right)\\
&= \mu_0(s)+\sum_{s'\in\mathcal{S}}\sum_{a'\in\mathcal{A}}P(s|s',a')\cdot\sum_{t=0}^{\infty}\gamma^{t+1}\mathbb{P}\left(S_t=s',A_t=a'|\mu_0,\pi,P\right)\\
&= \mu_0(s)+\sum_{s'\in\mathcal{S}}\sum_{a'\in\mathcal{A}}P(s|s',a')\cdot\gamma\sum_{t=0}^{\infty}\gamma^t\mathbb{P}\left(S_t=s',A_t=a'|\mu_0,\pi,P\right)\\
&= \mu_0(s)+\sum_{s'\in\mathcal{S}}\sum_{a'\in\mathcal{A}}P(s|s',a')\cdot\gamma\nu^{\pi}_{\mu_0}(s',a')\\
&= \mu_0(s)+\gamma\sum_{s'\in\mathcal{S}}\sum_{a'\in\mathcal{A}}P(s|s',a')\nu^{\pi}_{\mu_0}(s',a')
\end{aligned}
$$

1.2

Given the definition of the state occupancy measure:

$$\rho^{\pi}_{\mu_0}(s)=\sum_{t=0}^{\infty}\gamma^t\mathbb{P}(S_t=s|\mu_0,\pi,P)$$

We can marginalize the probability over all actions in $\mathcal{A}$ and rearrange the order of the summation:

$$\begin{aligned}
\rho^{\pi}_{\mu_0}(s) &= \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(S_t = s | \mu_0, \pi, P) \\
&= \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(S_t = s | \mu_0, \pi, P) \\
&= \sum_{t=0}^{\infty} \gamma^t \sum_{a \in \mathcal{A}} \mathbb{P}(S_t = s, A_t = a | \mu_0, \pi, P) \\
&= \sum_{a \in \mathcal{A}} \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(S_t = s, A_t = a | \mu_0, \pi, P) \\
\rho^{\pi}_{\mu_0}(s) &= \sum_{a \in \mathcal{A}} \nu^{\pi}_{\mu_0}(s, a) \qquad \blacksquare
\end{aligned}$$

Given the definition of the state-action occupancy measure, we can use the definition of conditional probability to construct a product of two probabilities, substitute in the policy function using its definition, and finally use the definition of the state occupancy measure:

$$\begin{aligned}
\nu^{\pi}_{\mu_0}(s, a) &= \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(S_t = s, A_t = a | \mu_0, \pi, P) \\
&= \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(S_t = s | \mu_0, \pi, P) \mathbb{P}(A_t = a | S_t = s, \mu_0, \pi, P) \\
&= \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(S_t = s | \mu_0, \pi, P) \pi(a | s) \\
\nu^{\pi}_{\mu_0}(s, a) &= \rho^{\pi}_{\mu_0}(s) \pi(a | s) \qquad \blacksquare
\end{aligned}$$

1.3

We just showed in 1.2 that

$$\rho^{\pi}_{\mu_0}(s) = \sum_{a \in \mathcal{A}} \nu^{\pi}_{\mu_0}(s, a)$$

And in 1.1 that

$$\sum_{a \in \mathcal{A}} \nu^{\pi}_{\mu_0}(s, a) = \mu_0(s) + \gamma \sum_{s' \in \mathcal{S}} \sum_{a' \in \mathcal{A}} P(s | s', a') \nu^{\pi}_{\mu_0}(s', a')$$

So by the transitive property:

$$\rho^{\pi}_{\mu_0}(s) = \mu_0(s) + \gamma \sum_{s' \in \mathcal{S}} \sum_{a' \in \mathcal{A}} P(s|s', a') \nu^{\pi}_{\mu_0}(s', a')$$

To convert this into vector form, the first μ_0 term in the sum is easy to take care of as it is added separately, so we can just form the equivalent vector:

$$\boldsymbol{\mu_0} = \begin{bmatrix} \mu_0(s_1) \\ \mu_0(s_2) \\ \vdots \\ \mu_0(s_{|\mathcal{S}|}) \end{bmatrix}$$

Next, we need the sum across all states and actions of the product of the transition function with the state-action occupancy measure. However, the transition function for each iteration of the summation needs to be evaluated for the state-action pair given by the summation transitioning to the state corresponding to the current index in the resulting vector. The state-action occupancy measure needs to be evaluated for the current state-action pair given by the summation. We also showed in 1.2 that

$$\nu^{\pi}_{\mu_0}(s, a) = \rho^{\pi}_{\mu_0}(s)\pi(a|s)$$

So the summation term in question can be rewritten as:

$$\gamma \sum_{s' \in \mathcal{S}} \sum_{a' \in \mathcal{A}} P(s|s', a') \rho^{\pi}_{\mu_0}(s') \pi(a'|s')$$

If we rewrite the P and π terms in terms of the probabilities they express and expand these probabilities using the definition of conditional probability, we get:

$$
\begin{aligned}
&\gamma \sum_{s' \in \mathcal{S}} \sum_{a' \in \mathcal{A}} \mathbb{P}(S_t = s | S_{t-1} = s', A_{t-1} = a') \mathbb{P}(A_{t-1} = a' | S_{t-1} = s') \rho^{\pi}_{\mu_0}(s') \\
&= \gamma \sum_{s' \in \mathcal{S}} \sum_{a' \in \mathcal{A}} \frac{\mathbb{P}(S_t = s, S_{t-1} = s', A_{t-1} = a')}{\mathbb{P}(S_{t-1} = s', A_{t-1} = a')} \frac{\mathbb{P}(A_{t-1} = a', S_{t-1} = s')}{\mathbb{P}(S_{t-1} = s')} \rho^{\pi}_{\mu_0}(s') \\
&= \gamma \sum_{s' \in \mathcal{S}} \sum_{a' \in \mathcal{A}} \frac{\mathbb{P}(S_t = s, S_{t-1} = s', A_{t-1} = a')}{\mathbb{P}(S_{t-1} = s')} \rho^{\pi}_{\mu_0}(s') \\
&= \gamma \sum_{s' \in \mathcal{S}} \rho^{\pi}_{\mu_0}(s') \sum_{a' \in \mathcal{A}} \frac{\mathbb{P}(S_t = s, S_{t-1} = s', A_{t-1} = a')}{\mathbb{P}(S_{t-1} = s')} \\
&= \gamma \sum_{s' \in \mathcal{S}} \rho^{\pi}_{\mu_0}(s') \frac{\mathbb{P}(S_t = s, S_{t-1} = s')}{\mathbb{P}(S_{t-1} = s')} \\
&= \gamma \sum_{s' \in \mathcal{S}} \rho^{\pi}_{\mu_0}(s') \mathbb{P}(S_t = s | S_{t-1} = s') \\
&= \gamma \sum_{s' \in \mathcal{S}} \rho^{\pi}_{\mu_0}(s') P(s | s')
\end{aligned}
$$

And we can create this sum using matrix multiplication. Since $\boldsymbol{P}^{\pi}_{ij} = P(s_j | s_i)$, this means that a single column of $\boldsymbol{P}^{\pi}$ corresponds to the different states s' could be. Standard matrix multiplication of $\boldsymbol{P}^{\pi}$ against some column vector would multiply rows against the vector, though. So, if we want to multiply the columns to achieve a summation of the product of $P(s|s')$ with $\rho^{\pi}_{\mu_0}(s')$ across $s' \in \mathcal{S}$ rather than $s \in \mathcal{S}$, we need to transpose $\boldsymbol{P}^{\pi}$ first and then matrix-multiply it against ${\boldsymbol{\rho}_{\boldsymbol{\mu_0}}}^{\pi}$:

$$
\begin{aligned}
\gamma \sum_{s' \in \mathcal{S}} \rho^{\pi}_{\mu_0}(s') P(s | s') &= \gamma \begin{bmatrix} P^{\pi}(s_1|s_1) & P^{\pi}(s_2|s_1) & \cdots & P^{\pi}(s_{|\mathcal{S}|}|s_1) \\ P^{\pi}(s_1|s_2) & P^{\pi}(s_2|s_2) & \cdots & P^{\pi}(s_{|\mathcal{S}|}|s_2) \\ \vdots & \vdots & \ddots & \vdots \\ P^{\pi}(s_1|s_{|\mathcal{S}|}) & P^{\pi}(s_2|s_{|\mathcal{S}|}) & \cdots & P^{\pi}(s_{|\mathcal{S}|}|s_{|\mathcal{S}|}) \end{bmatrix}^T \begin{bmatrix} \rho^{\pi}_{\mu_0}(s_1) \\ \rho^{\pi}_{\mu_0}(s_2) \\ \vdots \\ \rho^{\pi}_{\mu_0}(s_{|\mathcal{S}|}) \end{bmatrix} \\
&= \gamma \boldsymbol{P}^{\pi T} \boldsymbol{\rho}^{\boldsymbol{\pi}}_{\boldsymbol{\mu_0}}
\end{aligned}
$$

So finally, substituting this into the original sum:

$$
\boldsymbol{\rho}^{\boldsymbol{\pi}}_{\boldsymbol{\mu_0}} = \boldsymbol{\mu_0} + \gamma \boldsymbol{P}^{\pi T} \boldsymbol{\rho}^{\boldsymbol{\pi}}_{\boldsymbol{\mu_0}}
$$

1.4

$$\begin{aligned}
\boldsymbol{\rho}^{\boldsymbol{\pi}}_{\boldsymbol{\mu_0}} &= \boldsymbol{\mu_0} + \gamma \boldsymbol{P}^{\pi T} \boldsymbol{\rho}^{\boldsymbol{\pi}}_{\boldsymbol{\mu_0}} \\
\boldsymbol{\rho}^{\boldsymbol{\pi}}_{\boldsymbol{\mu_0}} - \gamma \boldsymbol{P}^{\pi T} \boldsymbol{\rho}^{\boldsymbol{\pi}}_{\boldsymbol{\mu_0}} &= \boldsymbol{\mu_0} \\
\left(\boldsymbol{I} - \gamma \boldsymbol{P}^{\pi T}\right) \boldsymbol{\rho}^{\boldsymbol{\pi}}_{\boldsymbol{\mu_0}} &= \boldsymbol{\mu_0} \\
\boldsymbol{\rho}^{\boldsymbol{\pi}}_{\boldsymbol{\mu_0}} &= \left(\boldsymbol{I} - \gamma \boldsymbol{P}^{\pi T}\right)^{-1} \boldsymbol{\mu_0}
\end{aligned}$$

The inverted quantity in the final line takes on the same form as the quantity proven to be invertible in Problem 1 of Homework 2: $\boldsymbol{I} - \gamma \boldsymbol{P}^{\pi}$. As a reminder, this quantity was shown to be invertible by an analysis of the eigenvalues. We showed that all the eigenvalues of $\boldsymbol{P}^{\pi}$ were nonzero by considering a theoretical eigenvalue to be greater than 1, which then proved to be impossible. From there, we showed that the eigenvalues of the entire quantity would necessarily be nonzero, proving that the quantity was invertible.

Now, if $(\boldsymbol{I} - \gamma \boldsymbol{P}^{\pi})$ is invertible, then so is $(\boldsymbol{I} - \gamma \boldsymbol{P}^{\pi T})$ since a matrix and its transpose always have the same eigenvalues, so the same argument applies to prove that this quantity is invertible.

To say that $(\boldsymbol{I} - \gamma \boldsymbol{P}^{\pi T})$ is invertible is to say that the equation $\boldsymbol{x}(\boldsymbol{I} - \gamma \boldsymbol{P}^{\pi T}) = \boldsymbol{b}$ has a unique for $\boldsymbol{x}$ given some arbitrary $\boldsymbol{b}$. To show this, consider some matrix $\boldsymbol{A}$ that is invertible:

$$\begin{aligned}
\boldsymbol{A}\boldsymbol{x} &= \boldsymbol{b} \\
\boldsymbol{A}^{-1}\boldsymbol{A}\boldsymbol{x} &= \boldsymbol{A}^{-1}\boldsymbol{b} \\
\boldsymbol{I}\boldsymbol{x} &= \boldsymbol{A}^{-1}\boldsymbol{b} \\
\boldsymbol{x} &= \boldsymbol{A}^{-1}\boldsymbol{b}
\end{aligned}$$

$\boldsymbol{A}^{-1}\boldsymbol{b}$ always yields a single vector, which is the unique solution to the equation. Our equation

$$\left(\boldsymbol{I} - \gamma \boldsymbol{P}^{\pi T}\right) \boldsymbol{\rho}^{\boldsymbol{\pi}}_{\boldsymbol{\mu_0}} = \boldsymbol{\mu_0}$$

takes on this exact form, such that

$$\boldsymbol{\rho}^{\boldsymbol{\pi}}_{\boldsymbol{\mu_0}} = \left(\boldsymbol{I} - \gamma \boldsymbol{P}^{\pi T}\right)^{-1} \boldsymbol{\mu_0}$$

is the unique solution to $\rho^{\pi}_{\mu_0}$.

1.5

Using the definition of $\nu^{\pi}_{\mu_0}$ from 1.2, we can rearrange to solve for π, then substitute in the definition of $\rho^{\pi}_{\mu_0}(s)$ from 1.2:

$$\begin{aligned}
\nu^{\pi}_{\mu_0}(s,a) &= \rho^{\pi}_{\mu_0}(s)\pi(a|s) \\
\pi(a|s) &= \frac{\nu^{\pi}_{\mu_0}(s,a)}{\rho^{\pi}_{\mu_0}(s)} \\
\pi(a|s) &= \frac{\nu^{\pi}_{\mu_0}(s,a)}{\sum_{a' \in \mathcal{A}} \nu^{\pi}_{\mu_0}(s,a')}
\end{aligned}$$

This is the first part of the piecewise equation, and this value is undefined if the denominator is zero, so we can just assign an arbitrary policy anywhere this is the case. An optimal, stationary, stochastic policy will comply with this construction because of the relationship between the occupancy measures and the policy function proven in 1.2.

1.6

It is given that

$$V^{\pi}(\mu) = \mathbb{E}_{S\sim\mu}\left[V^{\pi}(S)\right] = \sum_{s\in\mathcal{S}} \mu(s)V^{\pi}(s)$$

So, substituting in μ_0:

$$\begin{aligned}
V^{\pi}(\mu_0) &= \mathbb{E}_{S\sim\mu_0}\left[V^{\pi}(S)\right] = \sum_{s\in\mathcal{S}} \mu_0(s)V^{\pi}(s) \\
V^{\pi}(\mu_0) &= \sum_{s\in\mathcal{S}} \mu_0(s)\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t,a_t)|s_0 = s\right] \\
V^{\pi}(\mu_0) &= \sum_{s\in\mathcal{S}} \mathbb{P}(s_0 = s) \sum_{a\in\mathcal{A}} \sum_{s\in\mathcal{S}} \left[\mathbb{P}(s_0 = s|s_0 = s)\mathbb{P}(A_0 = a|s_0 = s)\sum_{t=0}^{\infty} \gamma^t R(s_t,a_t)\right] \\
V^{\pi}(\mu_0) &= \sum_{s\in\mathcal{S}} \mathbb{P}(s_0 = s) \sum_{a\in\mathcal{A}} \sum_{s\in\mathcal{S}} \left[1\cdot\pi(a|s)\sum_{t=0}^{\infty} \gamma^t R(s_t,a_t)|s_0 = s\right] \\
V^{\pi}(\mu_0) &= \sum_{s\in\mathcal{S}} \sum_{a\in\mathcal{A}} \nu^{\pi}_{\mu_0}(s,a)R(s,a)
\end{aligned}$$

1.7

The role of the objective function in the Dual LP for policy optimization is to maximize the sum of the product of the state-action occupancy measure across all states and actions, but we just showed above that this sum is equal to the value function. So, **the objective function maximizes the value function across all valid states and actions**. The constraint on this function is exactly the same equation we proved in 1.1, and it serves to restrain the state-action occupancy measure to valid values based on its definition in terms of the state transition probabilities and the initial state distribution. Combining these two, we can find an optimal policy.

Problem 2

2.1

If $|R(s,a)| \leq R_{max}$ and

$$G_i^s = \sum_{t=0}^{|\tau_i^s|} \gamma^t R(S_{t,i}^s, S_{t,i}^s)$$

then in the case where R is greatest, i.e. $R = R_{max}$,

$$G_i^s = \sum_{t=0}^{|\tau_i^s|} \gamma^t R_{max} = R_{max} \frac{1-\gamma^{|\tau_i^s|+1}}{1-\gamma}$$

and in the case where R is least, i.e. $R = -R_{max}$,

$$G_i^s = \sum_{t=0}^{|\tau_i^s|} -\gamma^t R_{max} = -R_{max} \frac{1-\gamma^{|\tau_i^s|+1}}{1-\gamma}$$

Therefore:

$$-R_{max} \frac{1-\gamma^{|\tau_i^s|+1}}{1-\gamma} \leq G_i^s \leq R_{max} \frac{1-\gamma^{|\tau_i^s|+1}}{1-\gamma}$$

Note that τ_i^s is not a known quantity until the trajectory is sampled, so we could use an even more conservative bound by assuming an infinite number of timesteps taken in the trajectory, in which we have:

$$\frac{-R_{max}}{1-\gamma} \leq G_i^s \leq \frac{R_{max}}{1-\gamma}$$

2.2

$$E(s) = \sum_{i=1}^{N^s} G_i^s$$

$$\begin{aligned}
\mathbb{E}_{A_{t\sim\pi}, S_{t+1}\sim P}[E(s)] &= \mathbb{E}_{A_{t\sim\pi}, S_{t+1}\sim P}\left[\sum_{i=1}^{N^s} G_i^s\right] \\
&= \sum_{i=1}^{N^s} \mathbb{E}_{A_{t\sim\pi}, S_{t+1}\sim P}\left[G_i^s\right] \\
&= \sum_{i=1}^{N^s} V^\pi(s)
\end{aligned}$$

2.3

Since we were able to bound G_i^s on both sides, and since $E(s) = \sum_{i=1}^{N^s} G_i^s$ is a sum of that bound variable, we can apply Hoeffding's inequality to bound the probability that $E(s)$ deviates from its expected value:

$$\mathbb{P}\left(|E(s) - \mathbb{E}[E(s)]| \geq \varepsilon\right) \leq 2\exp\left(-\frac{2\varepsilon^2}{\sum_{i=1}^{N^s}\left(2R_{max}\frac{1-\gamma^{|\tau_i^s|+1}}{1-\gamma}\right)^2}\right) = 2\exp\left(-\frac{\varepsilon^2}{\sum_{i=1}^{N^s} 2\left(R_{max}\frac{1-\gamma^{|\tau_i^s|+1}}{1-\gamma}\right)^2}\right)$$

2.4

We can apply the corrolary to Hoeffding's inequality which allows us to bound the arithmetic mean of the summed term:

$$\mathbb{P}\left(|\bar{s}_t - E[\bar{s}_t] \geq \varepsilon|\right) \leq 2\exp\left(\frac{-2\varepsilon^2 t}{(b_i - a_i)^2}\right), \quad \forall\varepsilon \geq 0$$

If s_{N^s} was the sum in the previous step, then

$$\bar{s}_i = \frac{1}{N^s} \sum_{i=1}^{N^s} G_i^s = \hat{V}^\pi(s)$$

And then the expected value of $\bar{s}_{N^s}$ is:

$$\mathbb{E}\left[\frac{1}{N^s} \sum_{i=1}^{N^s} G_i^s\right] = \frac{1}{N^s} \sum_{i=1}^{N^s} \mathbb{E}\left[G_i^s\right] = \frac{1}{N^s} \sum_{i=1}^{N^s} V^\pi(s) = \frac{1}{N^s} N^s V^\pi(s) = V^\pi(s)$$

So we have $\bar{s}_i$ and $\mathbb{E}[\bar{s}_i]$ to use in the alternate form of Hoeffding's inequality:

$$\mathbb{P}\left(\left|\hat{V}^\pi(s) - V^\pi(s)\right| \geq \varepsilon'\right) \leq 2\exp\left(-\frac{2(\varepsilon')^2 N^s}{\left(2R_{max}\frac{1-\gamma^{|\tau_i^s|+1}}{1-\gamma}\right)^2}\right) = 2\exp\left(-\frac{(\varepsilon')^2 N^s}{2\left(R_{max}\frac{1-\gamma^{|\tau_i^s|+1}}{1-\gamma}\right)^2}\right), \quad \forall \varepsilon' \geq 0$$

2.5

As N^s decreases, the term inside the exponential in 2.4 becomes less negative, so the exponential term itself becomes greater. Therefore, the smallest N^s will yield the largest $\mathbb{P}\left(\left|\hat{V}^\pi(s) - V^\pi(s)\right| \geq \varepsilon'\right)$. In other words, N will yield the largest probability of all N^s, so we can use it to derive an upper bound on $\mathbb{P}\left(\|\hat{V}^\pi - V^\pi\|_\infty \geq \varepsilon'\right)$. We have already calculated the bound for the absolute difference at any given state, and the L-∞ norm will give us the maximum absolute difference across all states. We do not have a way to calculate which state will yield the maximum absolute difference, so we can instead calculate the total probability that the absolute difference will be greater than ε' for *any* state, which is the union of all the events, which we can bound using Boole's inequality:

$$\mathbb{P}\left(\|\hat{V}^\pi - V^\pi\|_\infty \geq \varepsilon'\right) \leq \mathbb{P}\left(\bigcup_{s\in\mathcal{S}} \left|\hat{V}^\pi(s) - V^\pi(s)\right| \geq \varepsilon'\right) \leq \sum_{s\in\mathcal{S}} \mathbb{P}\left(\left|\hat{V}^\pi(s) - V^\pi(s)\right| \geq \varepsilon'\right)$$

And we can bound this sum using the bound we determined in 2.4 since the sum of a bounded function must be less than or equal to the sum of its bound. We substitute N in for N^s since we already showed how it will yield the greatest value, which is desired to achieve an upper bound:

$$\mathbb{P}\left(\|\hat{V}^{\pi} - V^{\pi}\|_{\infty} \geq \varepsilon'\right) \leq \sum_{s \in \mathcal{S}} \mathbb{P}\left(\left|\hat{V}^{\pi}(s) - V^{\pi}(s)\right| \geq \varepsilon'\right) \leq \sum_{i=1}^{|\mathcal{S}|} 2\exp\left(-\frac{(\varepsilon')^2 N^s}{2\left(R_{max}\frac{1-\gamma^{|\tau_i^s|+1}}{1-\gamma}\right)^2}\right)$$

$$\sum_{i=1}^{|\mathcal{S}|} 2\exp\left(-\frac{(\varepsilon')^2 N^s}{2\left(R_{max}\frac{1-\gamma^{|\tau_i^s|+1}}{1-\gamma}\right)^2}\right) = 2|\mathcal{S}|\exp\left(-\frac{(\varepsilon')^2 N^s}{2\left(R_{max}\frac{1-\gamma^{|\tau_i^s|+1}}{1-\gamma}\right)^2}\right)$$

So finally:

$$\mathbb{P}\left(\|\hat{V}^{\pi} - V^{\pi}\|_{\infty} \geq \varepsilon'\right) \leq 2|\mathcal{S}|\exp\left(-\frac{(\varepsilon')^2 N^s}{2\left(R_{max}\frac{1-\gamma^{|\tau_i^s|+1}}{1-\gamma}\right)^2}\right), \quad \forall \varepsilon' > 0$$

Note, once again, that we could substitute 1 *in for* $1 - \gamma^{|\tau_i^s|+1}$ *to get an even more conservative bound if we want to eliminate* $|\tau|$ *from the equation.*

Problem 3

First, we define the trajectories given, and after this we apply the every-visit Monte Carlo method to estimate $Q^{\pi}(b, x)$ based on the collected sample trajectories:

```
import itertools
from enum import Enum, auto

import numpy as np
from numpy.typing import NDArray


# Declare spaces
class State(Enum):
    b = auto()
    c = auto()
    d = auto()


class Action(Enum):
```

```
    x = auto()
    y = auto()


class Trajectory:
    states: list[State]
    actions: list[Action]
    rewards: NDArray[np.int64]

    def __init__(self, states, actions, rewards) -> None:
        self.states = states
        self.actions = actions
        self.rewards = rewards


trajectories = [
    Trajectory(
        [State.d, State.b, State.d, State.c],
        [Action.x, Action.x, Action.y],
        np.array([1.5, -1, 2]),
    ),
    Trajectory(
        [State.b, State.d, State.b, State.b, State.d, State.d, State.c],
        [Action.x, Action.y, Action.y, Action.x, Action.x, Action.x],
        np.array([-1, 2, 0, -1, 1.5, 1.5]),
    ),
    Trajectory(
        [State.b, State.d, State.d, State.d, State.b, State.c],
        [Action.y, Action.x, Action.x, Action.y, Action.y],
        np.array([0, 1.5, 1.5, 2, 0]),
    ),
]

# Initialize vhat
nonterminal_states = [State.b, State.d]


def initilize_for_each_state_action_pair(val):
    return {
        (state, action): val
        for state, action in itertools.product(nonterminal_states, Action)
    }
```

```python
qhat = initilize_for_each_state_action_pair(0.0)

# Initialize \mathcal{G}(s, a)
returns = initilize_for_each_state_action_pair([])

# Initialize N(s, a)
num_visits = initilize_for_each_state_action_pair(0)

for traj in trajectories:
    # For each state-action pair... (excluding state c since we end at that state)
    for state, action in itertools.product(nonterminal_states, Action):
        pair = (state, action)

        # If the pair in question isn't in the trajectory at all, then don't
        # attempt to update qhat
        if pair not in zip(traj.states[:-1], traj.actions):
            continue

        # For each step in the trajectory...
        for t, (s, a, reward) in enumerate(
            zip(traj.states[:-1], traj.actions, traj.rewards)
        ):
            # Skip if the step does not match the state-action pair in question
            if s != state or a != action:
                continue

            # We aren't told about any discount, so the return is just the  sum
            # of the rewards from the current time forward
            ret = float(np.sum(traj.rewards[t:]))
            returns[pair].append(ret)
            num_visits[pair] += 1

        qhat[pair] = float((1 / num_visits[pair]) * np.sum(returns[pair]))

for k, v in qhat.items():
    print(f"Qhat({k[0].name}, {k[1].name}) = {v}")
```

```
Qhat(b, x) = 3.5
Qhat(b, y) = 8.666666666666666
Qhat(d, x) = 6.9
```

```
Qhat(d, y) = 12.166666666666666
```

The resulting $\hat{Q}^\pi$ is shown above, and we can see that

$$Q^\pi(b, x) \approx \hat{Q}^\pi(b, x) = 3.5$$

Problem 4

To model the environment, we can re-use some of our code from HW2. It encodes the board, the given policy, the state transition probabilities, and the reward function:

```
import itertools
from enum import Enum, StrEnum, auto

import numpy as np
from IPython.display import Markdown


class BoardSpace(StrEnum):
    NORMAL = "N"
    MOUNTAIN = "M"
    LIGHTNING = "L"
    TREASURE = "T"


class Action(StrEnum):
    UP = "U"
    RIGHT = "R"
    DOWN = "D"
    LEFT = "L"


width = 5

board = np.full([width, width], BoardSpace.NORMAL)

board[2, 1] = BoardSpace.MOUNTAIN
board[3, 1] = BoardSpace.MOUNTAIN
board[1, 3] = BoardSpace.MOUNTAIN
board[2, 3] = BoardSpace.LIGHTNING
board[4, 4] = BoardSpace.TREASURE
```

```
policy = np.flipud(
    np.array(
        [
            list("RRRRU"),
            list("LULLU"),
            list("UURRR"),
            list("UDDDU"),
            list("URRUU"),
        ]
    )
).T

gamma = 0.95


def i_1d(x, y):
    return np.ravel_multi_index([y, x], dims=[width, width])


def is_blocked(x, y):
    return (
        x < 0
        or y < 0
        or x >= width
        or y >= width
        or board[x, y] == BoardSpace.MOUNTAIN
    )


def get_trans_prob(x: int, y: int, a: Action):
    if x < 0 or x >= width or y < 0 or y >= width:
        raise RuntimeError("Invalid coordinates")

    if a not in Action:
        raise RuntimeError("Invalid action type")

    p_vec = np.zeros(width**2)
    i_state_1d = i_1d(x, y)

    if board[x, y] != BoardSpace.NORMAL:
        p_vec[i_state_1d] = 1
        return p_vec
```

```
    direction_coordinates = {
        Action.LEFT: [x - 1, y],
        Action.RIGHT: [x + 1, y],
        Action.UP: [x, y + 1],
        Action.DOWN: [x, y - 1],
    }

    for direction, (x_next, y_next) in direction_coordinates.items():
        prob = 0.85 if direction == a else 0.05
        if is_blocked(x_next, y_next):
            p_vec[i_state_1d] += prob
        else:
            p_vec[i_1d(x_next, y_next)] += prob

    if p_vec.sum() != 1:
        raise RuntimeError("Probability vector did not add to 1")

    return p_vec


# (0, 0), (1, 0), (2, 0) ... (3, 4), (4, 4)
each_state = [(x, y) for y in range(5) for x in range(5)]


def p_matrix(policy):
    p = np.zeros([width**2, width**2])

    for x, y in each_state:
        i = i_1d(x, y)
        p[i] = get_trans_prob(x, y, policy[x, y])

    return p


reward_fn = np.zeros(width * width, dtype=int)
reward_fn[i_1d(*np.argwhere(board == BoardSpace.LIGHTNING).squeeze())] = -1
reward_fn[i_1d(*np.argwhere(board == BoardSpace.TREASURE).squeeze())] = 1


def vecfmt(v):
    return Markdown(", ".join([f"{e:g}" for e in v]))
```

4.1

To ensure that each state is visited often, we need to generate trajectories using a wide range of initial states. We could use a uniform random distribution to select our initial state for the trajectories, but in order to guarantee that all states get visited at least some given number of times, we will simply loop over each state (that is not a mountain) and generate the same number of trajectories beginning at that state. This ensures that each state is visited at least that many times.

We attempt to determine our termination condition using the criterion derived in 2.4. We showed there that

$$\mathbb{P}\left(\left|\hat{V}^{\pi}(s) - V^{\pi}(s)\right| \geq \varepsilon'\right) \leq 2\exp\left(-\frac{(\varepsilon')^2 N^s}{2\left(R_{max}\frac{1-\gamma^{|\tau_i^s|+1}}{1-\gamma}\right)^2}\right), \quad \forall \varepsilon' \geq 0$$

We don't know how long the trajectories will be since they are random, so we can assume they are infinitely long, which will give us an even more conservative estimate:

$$\mathbb{P}\left(\left|\hat{V}^{\pi}(s) - V^{\pi}(s)\right| \geq \varepsilon'\right) \leq 2\exp\left(-\frac{(\varepsilon')^2 N^s}{2\left(R_{max}\frac{1}{1-\gamma}\right)^2}\right), \quad \forall \varepsilon' \geq 0$$

Now, we need to supply some values for the desired deviation between the sample mean and its expected value, as well as the probability with which we want to be within that bound. We can choose a probability of 0.95 and a ε of 0.01, and solve for N^s, the number of sample trajectories required to achieve this goal. The probability is stated such that it is the probability that the absolute difference is *not* within ε, so we need to use $1 - 0.95$ on the LHS of the inequality:

$$\begin{aligned}
1 - 0.95 &\leq 2\exp\left(-\frac{0.01^2 N^s}{2\left(\frac{1}{1-0.95}\right)^2}\right) \\
\frac{0.05}{2} &\leq \exp\left(-\frac{0.0001}{800}N^s\right) \\
N^s &\geq -8 \cdot 10^6 \ln\left(\frac{0.05}{2}\right) \\
N^s &\geq 2.95110e + 07
\end{aligned}$$

Unfortunately, it appears that this upper bound is *extremely* conservative in this case, and we don't have the computational power to perform this many iterations. Luckily, through experimentation, it is shown that the value function converges long before then, and we choose

5000 iterations on each initial state since the max difference between iterations tails to nearly zero by this point, as shown in the plots later.

```
terminal_spaces = [BoardSpace.LIGHTNING, BoardSpace.TREASURE]

p = p_matrix(policy)

rng = np.random.default_rng(seed=42109581092395879)


def sample_trajectory(initial_state, policy):
    states = []
    actions = []
    rewards = []

    state = initial_state

    while True:
        x, y = state
        action = policy[x, y]
        reward = reward_fn[i_1d(x, y)]

        states.append(state)
        actions.append(action)
        rewards.append(reward)

        if board[*state] in terminal_spaces:
            break

        state_1d = rng.choice(
            np.arange(width**2), p=p[i_1d(x=state[0], y=state[1])]
        )

        y, x = np.unravel_index(state_1d, [width, width])

        state = np.array([x, y])

    return (np.array(vals) for vals in [states, actions, rewards])


# FIXME
# n_samples = 1
n_samples = 5000
```

```
vhat_mc = np.zeros([n_samples, width**2])

# Initialize vhat(s), \mathcal{G}(s), N(s)
vhat = np.zeros([width, width], dtype=float)
returns = {state: [] for state in each_state}
num_visits = np.zeros([width, width], dtype=int)

i = 0

reachable_states = [
    state for state in each_state if board[*state] != BoardSpace.MOUNTAIN
]

# for i in tqdm(range(n_samples)):
for i in range(n_samples):
    for initial_state in reachable_states:
        # Generate a sample trajectory using the policy
        states, actions, rewards = sample_trajectory(initial_state, policy)

        # If the pair in question isn't in the trajectory at all, then don't
        # attempt to update vhat
        for state in np.unique(states, axis=0):

            # Obtain the time at which the state is first visited
            t_s = np.argwhere(np.all(states == state, axis=1)).flatten()[0]

            # Get a vector of gamma to the (t - t_s) power for the timesteps from
            # t_s onward in order to calculate the return G
            gammas = np.pow(gamma, np.arange(len(states) - t_s))

            # Calculate G
            ret = float(np.sum(gammas * rewards[t_s:]))

            state_tuple = (int(state[0]), int(state[1]))

            returns[state_tuple].append(ret)

            num_visits[*state] += 1

            # Incremental calculation of the sample mean
            vhat[*state] = float(
                vhat[*state] + (1 / num_visits[*state]) * (ret - vhat[*state])
```

```
            )

    vhat_mc[i] = vhat.flatten()

print(f"Number of visits at each state:")
print(np.flipud(num_visits.T))
print(f"\n\nValue function at each state:")
print(np.flipud(vhat.T))
```

```
Number of visits at each state:
[[26210 31106 36159 39469 90763]
 [21973     0 19237 13402 51283]
 [20057  7664  9973 31421 48748]
 [11849  8553     0     0 40375]
 [ 6105 13598 19828 22464 28447]]


Value function at each state:
[[ 0.60582014  0.65929762  0.70245292  0.84530922  1.        ]
 [ 0.27901997  0.         -1.         -0.72213236  0.82920572]
 [ 0.25644694  0.13901678  0.17820083  0.26848268  0.35374119]
 [ 0.2322188   0.13206814  0.          0.          0.33357532]
 [ 0.21404735  0.13216579  0.13622205  0.1446888   0.30725486]]
```

The value function for each state on the board is shown above, as well as the number of visits at each state.

4.2

We implement the one-step TD algorithm below. Similarly to before, we force the algorithm to perform the same number of iterations using each reachable state as the initial state, guaranteeing that each state is visited at least that number of times. Also similarly to before, we use 5000 iterations since the plots below show convergence by this point in time.

```
# Initialize vhat
vhat = np.zeros([width, width], dtype=float)

# FIXME
# n_iterations = 1
n_iterations = 5000
```

```
learning_rate = 0.1

p = p_matrix(policy)

vhat_td = np.zeros([n_iterations, width**2])

# for i in tqdm(range(n_iterations)):
for i in range(n_iterations):
    # Sample an initial state randomly
    for state_i in reachable_states:
        state = np.array(state_i)
        x, y = state
        while True:
            # Obtain the immediate reward
            # reward = reward_fn[i_1d(x, y)]

            if board[*state] in terminal_spaces:
                # Take one more action and receive the reward at the terminal
                # state. Regardless of action we will obtain the same reward for
                # a terminal state.
                reward = reward_fn[i_1d(x, y)]
                v_state = vhat[x, y]

                # There is no new state after the terminal state, so there
                # should be no new value function
                v_state_new = 0

                vhat[x, y] = v_state + learning_rate * (
                    reward + gamma * v_state_new - v_state
                )

                break

            # Obtain the next state according to the transition function under the
            # policy (the correct action was already chosen when the p matrix was
            # calculated, so there is no need to explicitly get it here)
            state_1d_new = rng.choice(np.arange(width**2), p=p[i_1d(x, y)])
            y_new, x_new = np.unravel_index(state_1d_new, [width, width])
            state_new = np.array([x_new, y_new])

            # Collect the reward for the action taken at the current state
            reward = reward_fn[i_1d(x, y)]
```

```
            v_state = vhat[x, y]
            v_state_new = vhat[x_new, y_new]

            vhat[x, y] = v_state + learning_rate * (
                reward + gamma * v_state_new - v_state
            )

            state = state_new.copy()
            x, y = state

    vhat_td[i] = vhat.flatten()

print(np.flipud(vhat.T))
```

```
[[ 0.55112879  0.58501187  0.67787821  0.81542504  1.        ]
 [ 0.29473442  0.         -1.         -0.60777898  0.68622546]
 [ 0.26789398  0.15779418  0.27437646  0.27391521  0.49807159]
 [ 0.23616165  0.13913206  0.          0.          0.35267564]
 [ 0.19758343  0.14738108  0.17558833  0.21298866  0.33512462]]
```

4.3

```
lightning_coord = np.argwhere(board == BoardSpace.LIGHTNING).squeeze()
treasure_coord = np.argwhere(board == BoardSpace.TREASURE).squeeze()

r = np.zeros(width * width + 1, dtype=int)
r[i_1d(*lightning_coord)] = -1
r[i_1d(*treasure_coord)] = 1

p = np.vstack([p, np.zeros(25)])
p = np.hstack([p, np.zeros(26).reshape(-1, 1)])
p[25, :] = 0
p[:, 25] = 0
p[25, 25] = 1
p[i_1d(*lightning_coord), :] = 0
p[i_1d(*treasure_coord), :] = 0
p[i_1d(*lightning_coord), 25] = 1
p[i_1d(*treasure_coord), 25] = 1
```

```
v = (np.linalg.inv(np.identity(width**2 + 1) - gamma * p) @ r)[:-1]


print("Analytical value function solution:")
with np.printoptions(precision=4):
    print(np.flipud(v.reshape([5, 5])))
```

```
Analytical value function solution:
[[ 0.601   0.657   0.701   0.8471  1.    ]
 [ 0.2811  0.     -1.     -0.7148  0.8298]
 [ 0.2572  0.1426  0.1897  0.2743  0.3544]
 [ 0.2354  0.1308  0.      0.      0.3322]
 [ 0.2169  0.1321  0.1353  0.1439  0.3039]]
```

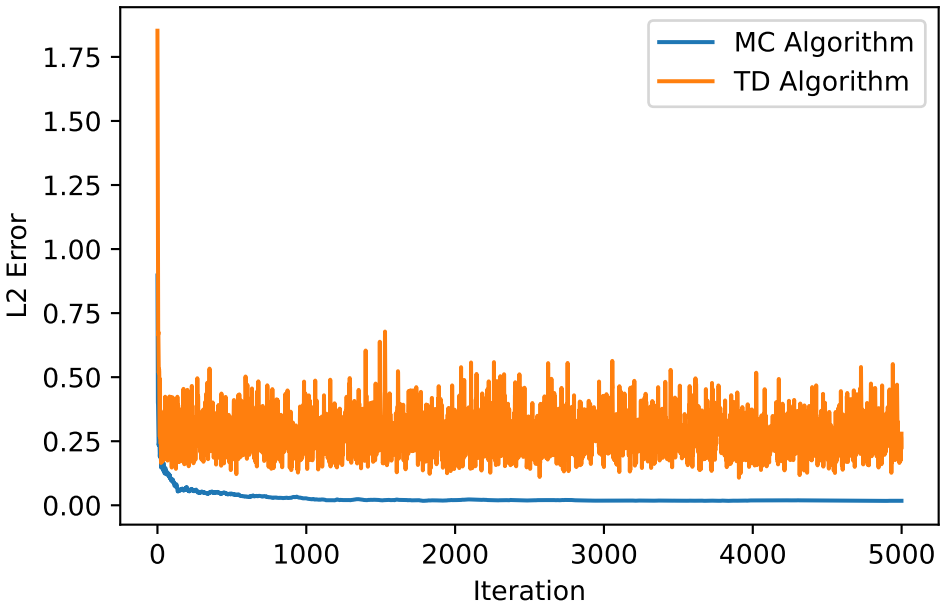
4.4

```
import matplotlib.pyplot as plt

v = v.reshape([5, 5]).T.flatten()

mc_err = np.linalg.norm(vhat_mc - v, axis=1)
td_err = np.linalg.norm(vhat_td - v, axis=1)

fig, ax = plt.subplots()
ax.plot(mc_err, label="MC Algorithm")
ax.plot(td_err, label="TD Algorithm")
ax.set_xlabel("Iteration")
ax.set_ylabel("L2 Error")
ax.legend()
```


MC Algorithm
TD Algorithm
L2 Error
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
1000
2000
3000
4000
5000
Iteration