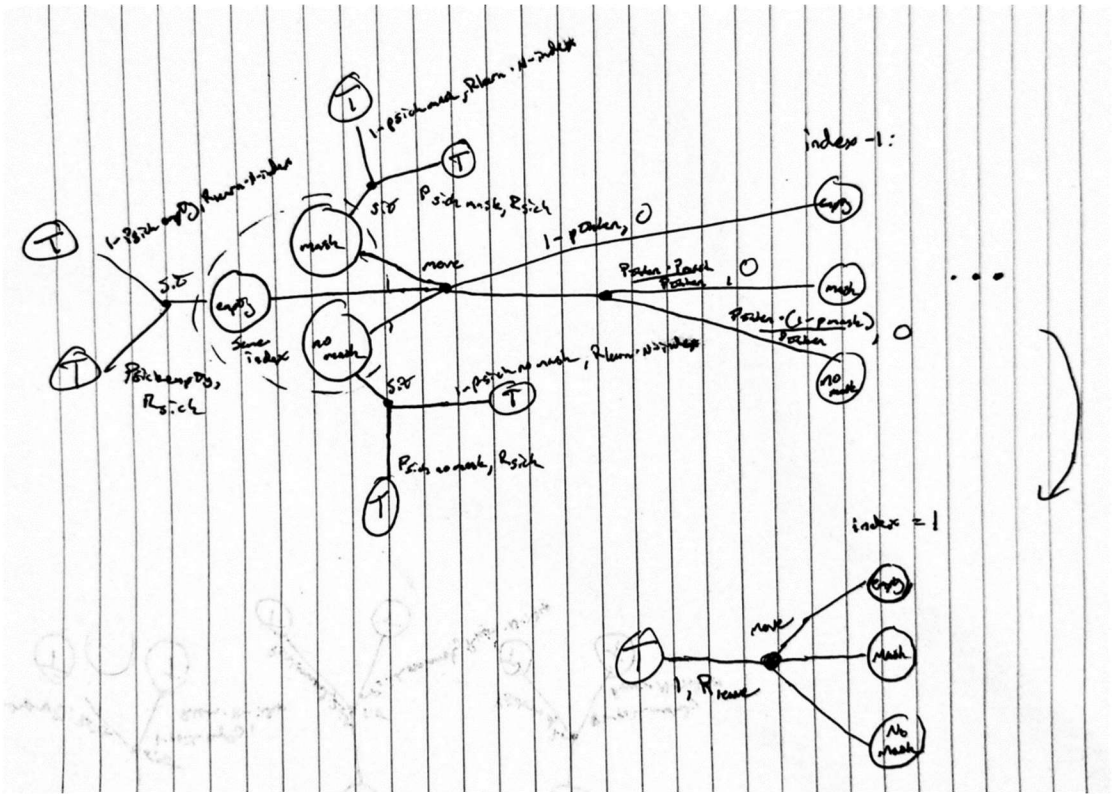
Dynamic Programming Report

Josh Bridges

The Problem

The reinforcement learning problem that is being examined today is finding an optimal policy for a student trying to learn in class. Based on who the student chooses to sit next to they have a chance of getting sick and getting reward R_{sick} on top of their reward for learning R_{learn}. If the student sits down, they will get a reward based on how close they are to the front row (R_{learn}*(N-rowIndex)). If the student moves onto the next seat from the front row, they exit the classroom and get R_{exit}.

The problem was modelled as an MDP by defining three states for each row based on the condition of the seat next to them.



1 Problem visualized as a Markov Decision Process (MDP)

The three states are set up for if the adjacent seat is empty, if the student seated there has a mask, or if the student has no mask. In the figure above, the three states on the left can each be seen mapping to two unique terminal states if the action is "sit"; these represent getting sick or being able to learn. The three states representing each row have a different probability of getting R_{learn} vs R_{sick} when the student chooses to sit, so those are modelled separately. These probabilities do not change when the student changes rows, so the model is the same for each row except for the row closest to the front at index one. The states at index one deterministically return R_{exit} if the student chooses to move in the row.

Code Design

My code for implementing value iteration is based around an N x 3 array of states representing the three states possible at each seat. The array is implemented as a list of size 3 vectors because it was simplest to instantiate and populate. The value function is arbitrarily set to zero on initialization. Since the target was value iteration and the process outputs a deterministic policy, the policy just stores an indicator of what action and has no need to store probabilities.

Each state has the value function and the policy for the state stored inside it. This allowed me to keep it simple and not need to create multiple N x 3 arrays for the different value and policy. With this implementation, the specification in the writeup that the functions need to take a value function as a parameter is satisfied by passing the array of states.

Results

Using the specifications from the writeup to run an instance of Value Iteration, my code produces the results seen on the right. Each row is separated by a line and the order of state in each row goes, from top to bottom: Empty, Taken-Mask, Taken-NoMask. The input parameters were N=12, R_{learn} = 2, R_{sick} = -100, R_{exit} = -100, p_{taken} = 0.5, p_{mask} = 0.5, $p_{sick\text{-}empty}$ = 0.01, p_{sick_masked} = 0.1, $p_{sick\_nomasked}$ = 0.5.

```
move    224.02674902997296
move    224.02674902997296
move    224.02674902997296

move    165.9457400222022
move    165.9457400222022
move    165.9457400222022

move    122.92277038681644
move    122.92277038681644
move    122.92277038681644

move    91.05390399023439
move    91.05390399023439
move    91.05390399023439

move    67.4473362890625
move    67.4473362890625
move    67.4473362890625

move    49.96098984375
move    49.96098984375
move    49.96098984375

move    37.008140625
move    37.008140625
move    37.008140625

move    27.4134375
move    27.4134375
move    27.4134375

move    20.306250000000002
move    20.306250000000002
move    20.306250000000002

sit     17.0
move    14.062500000000002
move    14.062500000000002

sit     19.0
sit     10.0
move    2.2500000000000018

sit     21.0
sit     12.0
sit     -28.0
```

2 Policy and Value Function for example case in writeup

For gamma and the threshold value, they were not specified so I used γ = 0.9 and threshold = 0.001.

These results make since to me because it is so easy to analyze the final row in this problem. Since the final row is followed by a terminal state in all situations, the value function is simply the greatest value between:

$$Q(s, move) = R_{exit} \quad \text{or}$$

$$Q(s, sit) = (p_{sick}) * R_{sick} + (R_{learn} * 11)$$

Q(s, move) is simply -100 so the Q(s, sit) has a higher value. Looking at V(Taken-NoMask) in figure 2 it is -28. We can solve this equation to see if it checks out

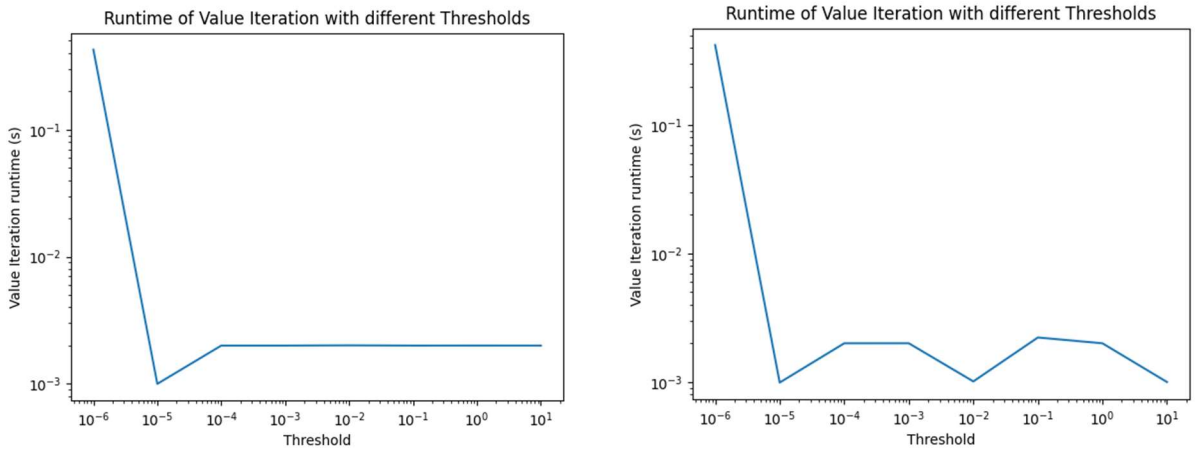
$$\begin{aligned} Q(s, sit) &= (p_{sick}) * R_{sick} + (R_{learn} * 11) = -28 \\ &= (p_{sick\_nm}) * R_{sick} + (R_{learn} * 11) \\ &= (0.5) * -100 + (2 * 11) \\ &= -50 + (0.5 * 11) = -28 \end{aligned}$$

The math checks out for this as well as the logic behind the value function. Many of the states have the same value along the row when they choose to move. This is because the probability model for what state a move action results in is independent of what state the action came from. As the row gets closer to the front row, the value of R_{exit} grows more prevalent in the value function and the value for move decreases. Eventually, in this

case, it gets surpassed in row 3 by the empty state but not the others. In row 2 move gets surpassed by sit in the masked state, and finally in the front row it is inherently worse than any of the sit options since the R_{exit} is -100.

Investigation

Since the example in the writeup did not specify a threshold or a gamma value I wanted to see how my selection of these values changed the aspects of value iteration.



3 loglog plots of Value Iteration runtime versus threshold value of the exact same parameter setup

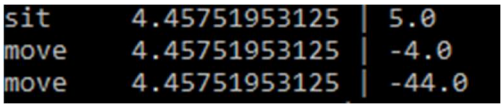
The first question that I wanted to see visualized was how the runtime of this value iteration changes based on the threshold value. For this analysis, I had to increase the number of states in order to get a significant sampling of time. I used 250 states in order to get a value greater than zero for the runtime. I noticed that the runtime fluctuates since the plots above are run using the exact same parameters. This is probably a factor of my local machine, however it is worth noticing that a threshold of 0.000001 or 10^{-6} results in a much longer runtime than larger thresholds. With such a large amount of states, the value function ends up being much larger. As a result, while the larger amounts of states are necessary for the runtime analysis to work, it probably increases the runtime significantly for low thresholds since the fluctuation of larger values will cause more iterations.

Since the "sit" action leads immediately to a terminal state, gamma only shows up in calculations of Q(s, move). So to look at the effect of the gamma on each iteration I sampled some of the values from the second to last states. Using gammas: [0, 0.1, 0.25, 0.5, 0.75, 0.9, 1], I looked at the Q(s, move) of the seat in row 2 to be able to see how it effects the values since row 1 does not use gamma by definition. The Q(s, move) increase as the gamma values increases which tracks because as seen by figure 2, the few states which have sit as a policy show an upward trend in the Q(s, sit) as they approach row 1. Gamma is a representation of how much relevance the future states have on current Q(s, move) values, therefore higher gamma's should

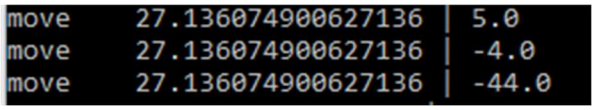
```
[ 0.0, 0.0, 0.0]
[ 0.25, 0.25, 0.25]
[ 0.625, 0.625, 0.625]
[ 1.25, 1.25, 1.25]
[ 1.875, 1.875, 1.875]
[ 2.2500000000000018, 2.2500000000000018, 2.2500000000000018]
[ 2.5, 2.5, 2.5]
```

4 Q(s, move) of states for seat on row 2 with changing gamma values

mean a higher Q(s, move) in this case. This can be seen to have an effect in Figure 5 and 6 with the same row, confirmed by the same Q(s, sit) values, having a different policy for the empty state (top row). The gamma cause the Q(s, move) to grow larger than the Q(s, sit) enough to actively change the policy.



6 Policy and Q(s, move) | Q(s, sit) of a row with gamma = 0.5



5 Policy and Q(s, move) | Q(s, sit) of the same row with gamma = 0.75

Conclusion

This project managed to implement dynamic programming on the student learning problem. The code uses value iteration to approximate the returns of all the states using the max of Q(s, move) and Q(s, sit). Then it evaluates and produces a deterministic policy for each state to specify the best action at that state. The code performs this all very quickly and can perform value iteration with very small thresholds in seconds depending on the size of the classroom (number of states).

I learned a lot from this project about how value iteration and the Bellman equations act around terminal states. The DP problems covered in class did not focus on terminal states so figuring out how that factors into these calculations was interesting and a facet of this project I had not thought about before. I also got a better understanding of the gamma value and how it correlates to the Q(s, a) of state-action pairs in these situations.

In summary this was a cool insight into a particular example of DP using terminal states that provided clear insight into the way value iteration works.