Nolan Gutierrez
CSE 6369 - Reinforcement Learning Homework 2- Spring 2020 Due Date: Apr. 21 2020
1)

a) Each Markov Decision Process (MDP) is usually defined by a transition function, a reward function, a state space and an action space. To start off, we can first design the state space but, first, to do this we need to define the problem statement. In this scenario we have an agent in a 15 by 25 grid. The agent has to navigate to a fixed goal using only four actions which are turn left, turn right, move forward, and move backward. Turning left and right can change the orientation of the agent while moving forward and backward changed the position within the grid world. Together, these four actions affect the pose of the agent within the grid world. For this problem the agent can be in 375 different positions on the 15 by 25 grid, and the turning actions can only rotate the agent ninety degrees clockwise or counterclockwise. In totality, this means the agent can have 4 different orientations for each state, making the total number of possible states for this MDP 1500. Since there are only four possible actions, we know that the action space has an order of four.

   We can now look at the reward function. If the agent chose to enter a position in the grid world with an obstacle in it, then the agent will receive a reward of -100. If the agent arrives in the position occupied by the goal, then the agent will receive a reward of 100. All other states will receive a reward of 0. All together, the reward is defined as follows:

```
function reward:
    args:
        action - one of four actions
        successor state - hypothetical successor state from taking action
    returns: reward
    if position of successor state is equal to position of goal then return 100
    else if position of successor state is equal to position of an obstacle,
        return -100
    else: return 0
```

   The transition function of this MDP is defined as follows:

   - For the actions forward and backward, the agent has a .8 probability of moving to the desired cell while the probability of staying in the same pose is .2
   - For both the turning actions, the agent has a probability of staying in the current state as .1 and a probability of successfully turning as .9.

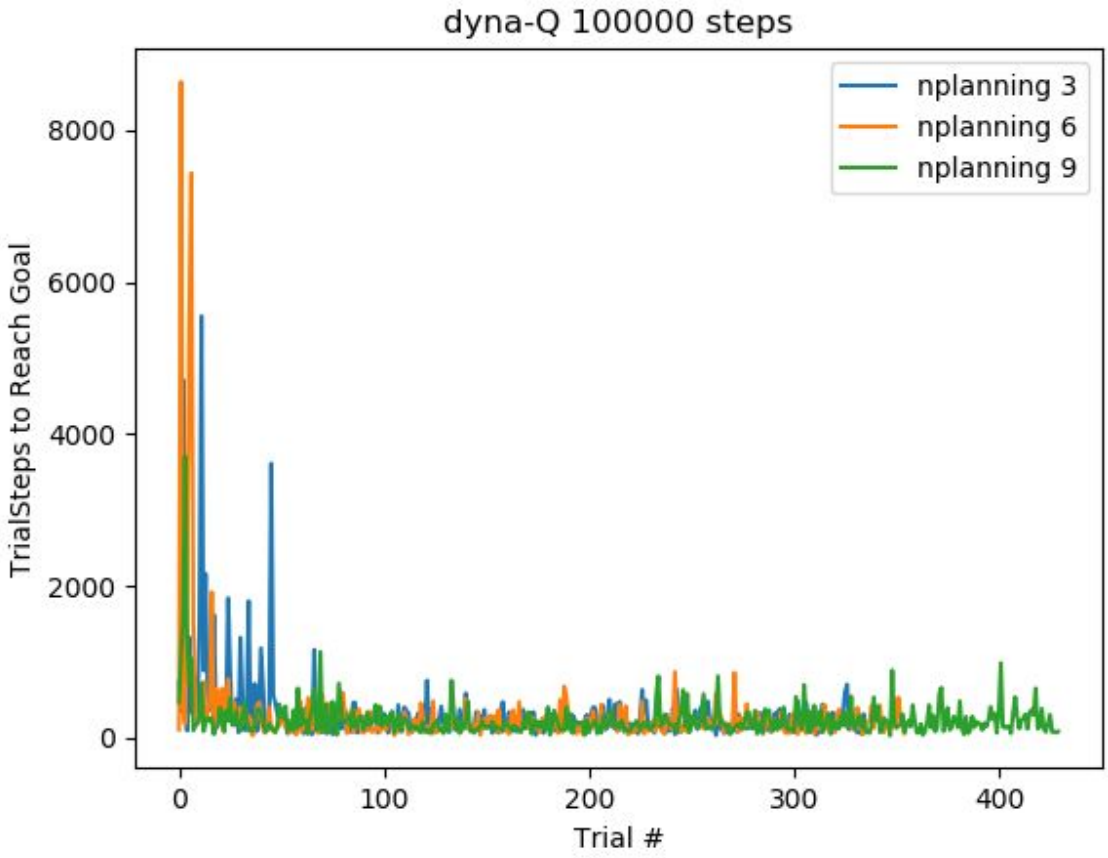
   To implement each state space, action space, reward function and transition function, python 3.7.6 was used. The implementation resulted in a class GridWorld which has its main function takeStep(). This function returns a successor state and reward from a given action and current state using the reward and transition functions defined here.

b) The Dyna-Q learner is implemented in python using the pseudocode on page 164 of the 2nd edition of the reinforcement learning book by Barto and Sutton. The Dyna-Q used an epsilon-greedy selection strategy where the value epsilon decays with the total

number of step sizes as follows: epsilon = 500 / total steps. The agent was ran for numerous different configurations, but the final agent was ran for 100k real world (grid world) steps. The results of this are described in the next section.

c) As the following graph shows, the agent shows the results of training a learning agent using the Dyna-Q algorithm to reach a fixed goal for a total of 100k steps. In the legend, the variable nplanning represents the number of planning or offline steps which are 3,6, and 9. We can note that agents with more offline steps per real world steps are able to finish more trials and reach a lower number of trial steps to reach the goal. The agent with 9 planning steps was able to complete dozens more trials within the allotted 100 thousand steps. Since our exploration strategy is defined as 500/ total steps, we know that initially the high variance of trial steps is due to a 100% exploration strategy. At the end of 100k steps we can calculate the epsilon value to be 500 / 100000 = .005. The hyperparameters used for this algorithm are discount factor set to .95, alpha set to .1, and the maximum number of experiences to keep in memory set at 500. The previously observed states used for offline learning are kept at the maximum value by randomly removing elements when the number of elements in experience is greater than the max.

It's unclear if the bit of variance still seen towards the end of each experiment would be reduced by increasing the total time allotted to each learner. As noted in the book for a similar grid world problem, the number of updates required in order to achieve the optimal solution for 1500 different states is in the ballpark of 10 to the power of 6 updates. Since the total number of updates seen in this experiment is still an order of magnitude less than that predicted number, we can safely assume that there is still more epistemic uncertainty contributing to the model's variance. We can also compare the performance of the three learning agents as shown below according to the time it takes to achieve a lower performance. If we judge the agents by the amount of real world updates that it takes for the agents to consistently obtain lower trial steps to reach the goal, then as shown below the agent with nplanning = 9 achieves lower variance for its low trial steps at an earlier time than the two other agents. We can see the next agent with the orange colored line follows in second place, and the agent with the lowest number of offline steps comes in last.

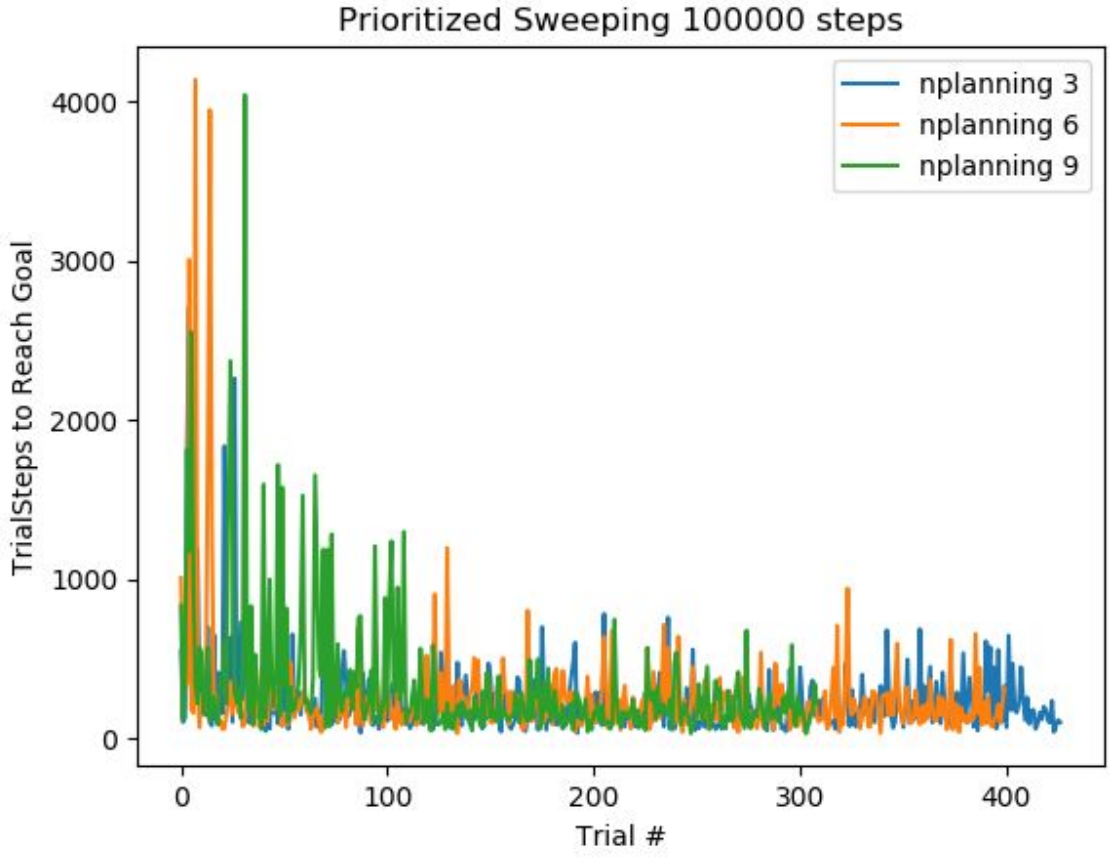


2)

a) A priority based system was implemented which bases its priority prediction difference magnitude as noted by Peng and Williams in their seminal paper “Efficient Learning and Planning Within the Dyna Framework” (1993). The implementation used for prioritized sweeping was based on the pseudocode on page 170 of the RL book by Barto and Sutton. There are a few implementation details which I will discuss. The exploration strategy that I used to obtain my results was a simple fixed epsilon-greedy strategy where epsilon is set to 0.01. While the prioritized sweeping method on page 170 is an on-policy algorithm, I decided to use fixed starting points instead of exploring starts which is the assumption of most on-policy algorithms. Using epsilon-greedy exploration allows the agents to sufficiently explore the state-action space. In order to obtain the predicted reward for the predecessor state-actions, the deterministic model developed in e) of the algorithm on page 170 was used. The hyper parameters of this algorithm are the following: alpha = 0.1, theta = .1, and discount factor = .95. The prioritized sweeping algorithm explores a much lower number of states while achieving similar results to the Dyna-Q algorithm as discussed next.

b) While the results from the Dyna-Q experiment makes intuitive sense, the results from this algorithm have little rationale. It was expected for the agent with the most number of planning steps to complete more trials in the same amount of time as the other two

agents. This is possibly due to the extra high variance towards the beginning of the experiment for this learner. The agent seems to have used most of its steps early on. More study would have to be devoted to concluding why this learner has higher variance for the agents with more planning steps. An additional possibility is that the agent is highly overfitting the data in some way. One additional thing to note is that while the agent with the most planning completed the fewer trials, a superficial qualitative analysis of the results shows that the green line has lower peaks and thus lower variance for its later trials when compared to the peaks of the orange and blue lines. We also see that the orange line has lower peaks for its later trials when compared to the peaks of the blue line. If we only base our analysis on the variance of the peaks for each learner rather than the number of trials completed or which agent achieves satisfactory performance first, then we can conclude that, indeed, the agent with more planning steps has better performance.



c) In order to implement a trajectory-based criterion for picking off-line steps, I decided to use trajectory sampling where offline steps are computed on-policy. Accomplishing this used some of the already implemented functions used with the Dyna-Q learner. For example I used the length-maintained experience from the Dyna-Q learner in order to obtain a randomly initialized start state from which to start an on policy trajectory. From

this state the agent takes the best action according to the current Q-values. The resulting reward and successor state are obtained from the deterministic model as maintained with the Dyna-Q learner. The Q-value update is performed with the resulting current state, on-policy action choice, reward and successor state. In order to continue the current trajectory, the current state for the off-line planning is set to the successor state for the next off-line step. The hyperparameters are the same as with Dyna-Q, with alpha = .1, discount factor = .95 and the maximum number of experiences at 500.

While incredibly consistent and with low variance, this dyna-q with trajectory sampling does not have intuitive sense. For one, the agent with the lowest number of planning steps achieves consistently lower trail steps to reach the goal than the other agents. This agent also has variance indistinguishable from the other agents. It appears, however, that this agent has better performance for the same number of updates when compared to the other two agents and their respective results. This agent also has training time similar to the training time it took for the prioritized sweeping agent, making this solution apparently preferable to the others. More analysis would have to be done to assess why more offline steps appear to provide worse results.