Stanford CME 241 (Winter 2022) - Midterm Exam

Instructions:

- We trust that you will follow The Stanford Honor Code.
- You have about 48 hours to work on this test, and need to submit your answers by 9 am PT on Wednesday 2/9. However, the estimated amount of work for this test is about 6 hours, depending on your proficiency in typesetting mathematical notations and in writing code. There are 3 problems (with subproblems), with a total of 25 points.
- Include all of your writing, math formulas, code, graphs etc. into a single answers-document. Be sure to write your name and SUNET ID in your answers-document. You can do your work in LaTeX or Markdown or Jupyter or any other typesetting option, which should finally be converted to the single answers-document PDF to be submitted. Note: Code can be included in LaTeX using the *lstlisting* environment, and graphs can be included using *includegraphics*.
- Submit your answers-document on Gradescope. Please ensure you have access to Gradescope before starting the exam so there is ample time to correct the problem. The Gradescope assignments for this midterm are here.
- Do not share or upload your code or written work publicly, and make sure to include the code files as part of your submission on Gradescope
- Submit a PDF of the written portion of your work to the `Midterm` Assignment on Gradescope, submit the code files to the `Midterm - Code` assignment on Gradescope

Problems:

1. **Question 1: Gaps Value Iteration Optimization.** In class we introduced the idea of the gaps-based asynchronous Value Iteration algorithm. In this question, we will implement this algorithm for Finite MDPs and see how it can effect convergence time for an environment with sparse rewards compared to an environment with dense rewards.

   The algorithm works as follows

   1. Initialize the value function to zero for all states: $v[s] = 0 \ \forall s \in \mathcal{N}$
   2. Calculate the gaps for each state: $g[s] = |v[s] - \max_a \mathcal{R}(s,a) + \gamma \sum_{s'} \mathcal{P}(s,a,s') \cdot v(s')|$
   3. While there is some gap that exceeds a threshold
      - Select the state with the largest gap: $s_{max} = \arg\max_{s\in\mathcal{N}} g[s]$
      - Update the value function for s_{max}: $v[s_{max}] = \max_a \mathcal{R}(s_{max},a) + \gamma \sum_{s'} \mathcal{P}(s_{max},a,s') \cdot v(s')$
      - Update the gap for s_{max}: $g[s_{max}] = 0$
      - Update the gaps for each state which depends directly on s_{max}:

   $$\forall s \in D(s_{max}): \ g[s] = |v[s] - \max_a \mathcal{R}(s,a) + \gamma \sum_{s'} \mathcal{P}(s,a,s') \cdot v(s')|$$

   $$D(s) = \{s' \in S | \ \mathcal{P}(s',a,s) > 0 \text{ for some action } a\}$$

   4. Return v

   Please pull the latest copy of the *RL−Book* repository (i.e. run `$ git pull` on the master branch); we have provided the code skeleton for the problem in the *midterm*_2022 folder. We have included two utility files which you should not edit, but which you may want to read through to familiarize yourself with the code. We have also included a jupyter notebook which has the skeleton for the code you should implement.

   1. **grid_maze.py**
      - This file contains the class definitions for the GridMaze MDPs
      - You should note the difference in the reward functions between `GridMazeMDP_Dense` and `GridMazeMDP_Sparse`
      - You should not edit the code in this file
   2. **priority_q.py**
      - This file contains the implementation of a the priority queue you should use to store the gaps
      - You should not edit the code in this file
      - You will need to make use of the PriorityQueue class in your implementation. A PriorityQueue is an ordered queue which supports the following operations where n is the number of elements in the queue
        (a) isEmpty(self): check if the queue is empty – Runtime: $O(1)$
        (b) contains(self, element): check if the queue contains an element – Runtime: $O(1)$
        (c) peek(self): peek at the highest priority element in the queue – Runtime: $O(1)$
        (d) pop(self): remove and return the highest priority element in the queue – Runtime: $O(\log(n))$

   (e) insert(self, element, priority): insert an element into the queue with given priority – Runtime: $O(\log(n))$
   (f) update(self, element, new_priority): update the priority of an element in the queue – Runtime: $O(\log(n))$
   (g) delete(self, element): delete an element from the queue – Runtime: $O(\log(n))$
 - We have included examples of the usage of the PriorityQueue class in the implementation file.

3. **midterm-22-P1-Skeleton.ipynb**
   - This file is a jupyter notebook which has the code-skeleton which you should complete
   - You **should** edit the code in this file *where indicated*
   - to run the notebook,
     (a) cd into the RL-Book repository:
     `$ cd {your repo location}/RL-Book`
     (b) activate the environment corresponding to the class:
     `$ source {your venv location}/bin/activate`
     (c) run jupyter:
     `$ jupyter notebook`
     (d) on a web-browser navigate to `localhost:8888` (or whatever port number is indicated by the output from the previous command), then navigate to the `midterm_2022` folder
     (e) if you have trouble with running this, email Sven: `svenl@stanford.edu` directly

Question 1 Problems

1. **2 pts:** Implement the invert_transition_mapping function, this should give you the states whose value function directly depends on another state's value function. i.e. this function maps $s \to I(s)$ where $I(s) = \{s' \in S | \; P(s', a, s) > 0 \text{ for some } a\}$. This will help you update the gaps in the second step of the algorithm

2. **4 pts:** Implement "gaps_value_iteration" and the wrapper function "gaps_value_iteration_result" using the algorithm above. (*Hint:* ensure that you are not adding unnecessary elements to the queue). You may wish to look at the implementation of the corresponding functions in dynamic_programming.py

   Test the time for convergence of your algorithm for the SparseGridMaze MDP with $\gamma = 0.9$ and the DenseGridMaze MDP with $\gamma = 1$ and compare it to the convergence time for standard value iteration. Also ensure that the value functions produced by your code matches the value function produced by the dynamic programming library. **Note** we have provided code for you to carry out this step.

3. **1 pts:** Explain the difference between the convergence time for the four configurations (Gaps / Standard value iteration on the Dense / Sparse GridMaze).

2. **Question 2** Assume that we are in discrete time, and imagine that at every time t you are allowed to allocate your wealth into two assets (A_1, A_2) which pay you money back at time $t+1$. For each dollar invested in A_1 at time t you receive X_{t+1} dollars at time $t+1$; for each dollar invested in A_2 at time t you receive Y_{t+1} dollars at time $t+1$. The joint probability of (X_{t+1}, Y_{t+1}) is invariant in time and follows the following (infinite) Probability Mass Function:

$$(X_{t+1}, Y_{t+1}) \sim \begin{cases} (1,0) & \text{with probability } \frac{1}{2} \\ (0,2^i) & \text{with probability } \frac{1}{2^{i+1}} \quad \text{for } i \in \mathbb{Z}^+ \end{cases}$$

You have a investment horizon of T time-steps and your goal is to maximize the utility of your wealth at the end of T time-steps by dynamically re-balancing your portfolio at each time-step. Assume that you have log utility ($U(w) = \log(w)$). You must invest all of your wealth at each time across the two assets, so you can represent your portfolio at any time as

$$\begin{cases} W_t(1-\pi_t) & \text{invested in } A_1 \\ W_t\pi_t & \text{invested in } A_2 \end{cases}$$

You begin at time $t = 0$ with some initial wealth W_0, and the only changes in your wealth result from changes in the value of your portfolio, i.e.

$$W_{t+1} = W_t((1-\pi_t)X_{t+1} + \pi_t Y_{t+1})$$

1. **3 points:** Consider the case where $T = 1$, what is the optimal portfolio choice, π which maximizes the expected utility of wealth after one time-step, $E[U(W_1)]$ where

$$W_1 = W_0 * ((1-\pi)X_1 + \pi Y_1)$$

2. **6 points:** Prove that the optimal policy when you have to invest over $T > 1$ periods is the same as the optimal policy for $T = 1$ (i.e.) $\pi_t^* = \pi^*$.

3. **Question 3: Dice Rolling**

   Consider the following dice game. You start with N K-sided dice on the table, and no dice in your hand. The values on the dice faces are $\{1, 2, ..., K\}$. While you have dice remaining on the table, the game proceeds as follows:

   1. roll all the dice on the table
   2. select a nonempty subset of the dice on the table to move to your hand, the dice you move to your hand keep the value which they were just rolled. For example, if your hand is $\{1, 3\}$ and you roll $(2, 2, 3, 4)$ and you decide to move the dice with 3&4 to your hand, you will now have $\{1, 3, 3, 4\}$ in your hand.

   The game ends when you have no dice on the table left to roll. Your score for the game is then calculated as the sum of the values of dice in your hand if you have at least C 1's in your hand, and zero otherwise. For example, for $N = K = 4$ and $C = 2$, the score corresponding to a hand containing (1, 3, 1, 4) would be 9 while the score corresponding to a hand containing (4, 1, 3, 4) would be 0.

   Your goal is to maximize your score at the end of the game.

   - **4 points:** With proper mathematical notation, model this as a Finite MDP specifying the states, actions, rewards, state-transition probabilities and discount factor.
   - **5 points:** Implement this MDP in python. If you wish, you may use the code in the git repo that you forked at the start of the course (eg: FiniteMarkovDecisionProcess), but if you prefer, you can implement it from scratch or using code you have written for the course previously (whichever is more convenient for you). You should implement this for the general case, specifically your MDP implementation should take as parameters N, K, C.

     For $N = 6, K = 4, C = 1$, use the dynamic_programming.py library (or your own code if you chose not to implement it within the class library) to solve for the optimal value function, and present the following values,

     1. The expected score of the game playing optimally, calculate this using your code, *not* analytically
     2. The optimal action when rolling $\{1, 2, 2, 3, 3, 4\}$ on the first roll