Homework 3 - Blackjack

Addapted from Peeking Blackjack Stanford CS221

TA in Charge: Siyuan Wang

General Instructions

This assignment has a written part and a programming part.

a.  This icon means a written answer is expected in `blackjack.pdf`.

b.  This icon means you should write code in `submission.py`.

You should modify the code in `submission.py` between

```
# BEGIN_YOUR_CODE
```

and

```
# END_YOUR_CODE
```

but you can add other helper functions outside this block if you want. Do not make changes to files other than `submission.py`.

Your code will be evaluated on two types of test cases, **basic** and **hidden**, which you can see in `grader.py`. Basic tests, which are fully provided to you, do not stress your code with large inputs or tricky corner cases. Hidden tests are more complex and do stress your code. The inputs of hidden tests are provided in `grader.py`, but the correct outputs are not. To run all the tests, type

```
python grader.py
```

This will tell you only whether you passed the basic tests. On the hidden tests, the script will alert you if your code takes too long or crashes, but does not say whether you got the correct output. You can also run a single test (e.g., `3a-0-basic`) by typing

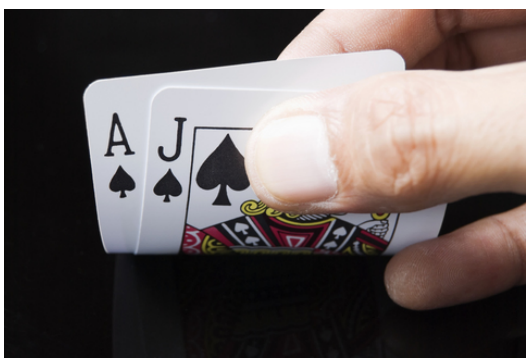
```
python grader.py 3a-0-basic
```

We strongly encourage you to read and understand the test cases, create your own test cases, and not just blindly run `grader.py`.

Submission

- Submit a zip file consisting of both **blackjack.pdf** and **submission.py** through E-learning before mid-night of May 27th, 2018.
- Name the zip file as **hw3-your-sid.zip**.
- For any question about this assignment, contact Siyuan Wang for more information..

Markov decision processes (MDPs) can be used to formalize uncertain situations. In this homework, you will implement algorithms to find the optimal policy in these situations. You will then formalize a modified version of Blackjack as an MDP, and apply your algorithm to find the optimal policy.



Problem 1: Value Iteration

In this problem, you will perform the value iteration updates manually on a very basic game just to solidify your intuitions about solving MDPs. The set of possible states in this game is {-2, -1, 0, 1, 2}. You start at state 0, and if you reach either -2 or 2, the game ends. At each state, you can take one of two actions: {-1, +1}.

If you're in state s and choose -1:

- You have an 80% chance of reaching the state $s-1$.
- You have a 20% chance of reaching the state $s+1$.

If you're in state s and choose +1:

- You have a 30% chance of reaching the state $s+1$.
- You have a 70% chance of reaching the state $s-1$.

If your action results in transitioning to state -2, then you receive a reward of 20. If your action results in transitioning to state 2, then your reward is 100. Otherwise, your reward is -5. Assume the discount factor γ is 1.

a. [3 points] Give the value of $V_{\text{opt}}(s)$ for each state s after 0, 1, and 2 iterations of value iteration. Iteration 0 just initializes all the values of V to 0. Terminal states do not have any optimal policies and take on a value of 0.

b. [3 points] What is the resulting optimal policy π_{opt} for all non-terminal states?

Problem 2: Transforming MDPs

Let's implement value iteration to compute the optimal policy on an arbitrary MDP. Later, we'll create the specific MDP for Blackjack.

a. [3 points] If we add noise to the transitions of an MDP, does the optimal value always get worse? Specifically, consider an MDP with reward function $\text{Reward}(s, a, s')$, states States, and transition function $T(s, a, s')$. Let's define a new MDP which is identical to the original, except that on each action, with probability $\frac{1}{2}$, we randomly jump to one of the states that we could have reached before with positive probability. Formally, this modified transition function is:

$$T'(s, a, s') = \frac{1}{2}T(s, a, s') + \frac{1}{2} \cdot \frac{1}{|\{s'' : T(s, a, s'') > 0\}|}.$$

Let V_1 be the optimal value function for the original MDP, and V_2 the optimal value function for the modified MDP. Is it always the case that $V_1(s_{\text{start}}) \geq V_2(s_{\text{start}})$? If so, prove it in `blackjack.pdf` and put `return None` for each of the code blocks. Otherwise, construct a counterexample by filling out `CounterexampleMDP`.

b. [3 points] Suppose we have an acyclic MDP (you will not visit a state a second time in this process). We could run value iteration, which would require multiple iterations. Briefly explain a more efficient algorithm that only requires one pass over all the (s, a, s') triples.

c. [3 points] Suppose we have an MDP with states States a discount factor $\gamma < 1$, but we have an MDP solver that only can solve MDPs with discount 1. How can leverage the MDP solver to solve the original MDP?

Let us define a new MDP with states $\text{States}' = \text{States} \cup \{o\}$, where o is a new state. Let's use the same actions ($\text{Actions}'(s) = \text{Actions}(s)$), but we need to keep the discount $\gamma' = 1$. Your job is to define new transition probabilities $T'(s, a, s')$ and rewards $\text{Reward}'(s, a, s')$ in terms of the old MDP such that the optimal values $V_{\text{opt}}(s)$ for all $s \in \text{States}$ are the equal under the original MDP and the new MDP.

Problem 3: Peeking Blackjack

Now that we have written general-purpose MDP algorithms, let's use them to play (a modified version of) Blackjack. For this problem, you will be creating an MDP to describe a modified version of Blackjack.

For our version of Blackjack, the deck can contain an arbitrary collection of cards with different values, each with a given multiplicity. For example, a standard deck would have card values $[1, 2, \ldots, 13]$ and multiplicity 4. You could also have a deck with card values $[1, 5, 20]$. The deck is shuffled (each permutation of the cards is equally likely).

The game occurs in a sequence of rounds. Each round, the player either (i) takes the next card from the top of the deck (costing nothing), (ii) peeks at the top card (costing `peekCost`, in which case the next round, that card will be drawn), or (iii) quits the game. (Note: it is not possible to peek twice in a row; if the player peeks twice in a row, then `succAndProbReward()` should return `[]`.)

The game continues until one of the following conditions becomes true:

- The player quits, in which case her reward is the sum of the cards in her hand.
- The player takes a card, and this leaves her with a sum that is strictly greater than the threshold, in which case her reward is 0.
- The deck runs out of cards, in which case it is as if she quits, and she gets a reward which is the sum of the cards in her hand.

In this problem, your state s will be represented as a triple:

```
(totalCardValueInHand, nextCardIndexIfPeeked, deckCardCounts)
```

As an example, assume the deck has card values $[1, 2, 3]$ with multiplicity 1, and the threshold is 4. Initially, the player has no cards, so her total is 0; this corresponds to state `(0, None, (1, 1, 1))`. At this point, she can take, peek, or quit.

- If she takes, the three possible successor states (each has $1/3$ probability) are

```
(1, None, (0, 1, 1))
(2, None, (1, 0, 1))
(3, None, (1, 1, 0))
```

She will receive reward 0 for reaching any of these states.

- If she instead peeks, the three possible successor states are

  ```
  (0, 0, (1, 1, 1))
  (0, 1, (1, 1, 1))
  (0, 2, (1, 1, 1))
  ```

  She will receive reward `-peekCost` to reach these states. From `(0, 0, (1, 1, 1))`, taking yields `(1, None, (0, 1, 1))` deterministically.
- If she quits, then the resulting state will be `(0, None, None)` (note setting the deck to `None` signifies the end of the game).

As another example, let's say her current state is `(3, None, (1, 1, 0))`.

- If she quits, the successor state will be `(3, None, None)`.
- If she takes, the successor states are `(3 + 1, None, (0, 1, 0))` or `(3 + 2, None, None)`. Note that in the second successor state, the deck is set to `None` to signify the game ended with a bust. You should also set the deck to `None` if the deck runs out of cards.

a. [10 points] Implement the game of Blackjack as an MDP by filling out the `succAndProbReward()` function of class `BlackjackMDP`.

b. [4 points] Let's say you're running a casino, and you're trying to design a deck to make people peek a lot. Assuming a fixed threshold of 20, and a peek cost of 1, design a deck where for at least 10% of states, the optimal policy is to peek. Fill out the function `peekingMDP()` to return an instance of `BlackjackMDP` where the optimal action is to peek in at least 10% of states.

Problem 4: Learning to Play Blackjack

So far, we've seen how MDP algorithms can take an MDP which describes the full dynamics of the game and return an optimal policy. But suppose you go into a casino, and no one tells you the rewards or the transitions. We will see how reinforcement learning can allow you to play the game and learn the rules at the same time!

a. [8 points] You will first implement a generic Q-learning algorithm `QLearningAlgorithm`, which is an instance of an `RLAlgorithm`. As discussed in class, reinforcement learning algorithms are capable of executing a policy while simultaneously improving their policy. Look in `simulate()`, in `util.py` to see how the `RLAlgorithm` will be used. In short, your `QLearningAlgorithm` will be run in a simulation of the MDP, and will alternately be asked for an action to perform in a given state (`QLearningAlgorithm.getAction`), and then be informed of the result of that action (`QLearningAlgorithm.incorporateFeedback`), so that it may learn better actions to perform in the future.

   We are using Q-learning with function approximation, which means $\hat{Q}_{\text{opt}}(s,a) = \mathbf{w} \cdot \phi(s,a)$, where in code, $\mathbf{w}$ is `self.weights`, ϕ is the `featureExtractor` function, and $\hat{Q}_{\text{opt}}$ is `self.getQ`.

   We have implemented `QLearningAlgorithm.getAction` as a simple ϵ-greedy policy. Your job is to implement `QLearningAlgorithm.incorporateFeedback()`, which should take an (s,a,r,s') tuple and update `self.weights` according to the standard Q-learning update.

b. [4 points] Call `simulate` using your algorithm and the `identityFeatureExtractor()` on the MDP `smallMDP`, with 30000 trials. Compare the policy learned in this case to the policy learned by value iteration. Don't forget to set the explorationProb of your Q-learning algorithm to 0 after learning the policy. How do the two policies compare (i.e., for how many states do they produce a different action)? Now run `simulate()` on `largeMDP`. How does the policy learned in this case compare to the policy learned by value iteration? What went wrong?

c. [5 points] To address the problems explored in the previous exercise, we incorporate domain knowledge to improve generalization. This way, the algorithm can use what it learned about some states to improve its prediction performance on other states. Implement `blackjackFeatureExtractor` as described in the code comments. Using this feature extractor, you should be able to get pretty close to the optimum on the `largeMDP`.

d. [4 points] Now let's explore the way in which value iteration responds to a change in the rules of the MDP. Run value iteration on `originalMDP` to compute an optimal policy. Then apply your policy to `newThresholdMDP` by calling `simulate` with `FixedRLAlgorithm`, instantiated using your computed policy. What reward do you get? What happens if you run Q learning on `newThresholdMDP` instead? Explain.