XCS221 Assignment 3 — Peeking Blackjack

Due Sunday, December 13 at 11:59pm PT.

Guidelines

1. These questions require thought, but do not require long answers. Please be as concise as possible.

2. If you have a question about this homework, we encourage you to post your question on our Slack channel, at
   http://xcs221-scpd.slack.com/

3. Familiarize yourself with the collaboration and honor code policy before starting work.

4. For the coding problems, you may not use any libraries except those defined in the provided started code. In particular, ML-specific libraries such as `scikit-learn` are not permitted.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.

- Submit the compiled PDF, **not** `submission.tex`.

- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. Your code will be autograded online using `src/grader.py`, which is provided for you in the `src/` subdirectory. You can also run this autograder on your local computer, although some of the tests will be skipped (since they require the instructor solution code for comparison).

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing `src/submission.py`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These unit tests will verify only that your code runs without errors on obvious test cases. These tests so not require the instructor solution code and can therefore be run on your local computer.

- `hidden`: These unit tests will verify that your code produces correct results on complex inputs and tricky corner cases. Since these tests require the instructor solution code to verify results, only the setup and inputs are provided. When you run the autograder locally, these test cases will run, but the results will not be verified by the autograder. When your run the autograder online, these tests will run and you will receive feedback on any errors that might occur.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

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

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Introduction

The search algorithms explored in the previous assignment work great when you know exactly the results of your actions. Unfortunately, the real world is not so predictable. One of the key aspects of an effective AI is the ability to reason in the face of uncertainty.

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.

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) **[4 points (Written)]** 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) **[4 points (Written)]** Based on $V_{\text{opt}}(s)$ after the second iteration, what is the corresponding optimal policy π_{opt} for all non-terminal states?

2. **Transforming MDPs**

Equipped with an understanding of a basic algorithm for computing optimal value functions in MDPs, let's gain intuition about the dynamics of MDPs which either carry some special structure, or are defined with respect to a different MDP.

(a) [**4 points (Coding)**] 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` in `submission.py`.

(b) [**4 points (Written)**] Suppose we have an acyclic MDP for which we want to find the optimal value at each node. We could run value iteration, which would require multiple iterations – but it would be nice to be more efficient for MDPs with this acyclic property. Briefly explain an algorithm that will allow us to compute V_{opt} for each node with only a single pass over all the (s, a, s') triples.

(c) [**4 points (Written)**]

Suppose we have an MDP with states S and a discount factor $\lambda < 1$, but we have an MDP solver that can only solve MDPs with discount factor of 1. How can we leverage the MDP solver to solve the original MDP?

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

Hint: If you're not sure how to approach this problem, go back to the first MDP lecture and read closely the slides on convergence.

3. **Peeking Blackjack**

Now that we have gotten a bit of practice with 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 states, actions, and rewards in this game.

For our version of Blackjack, the deck can contain an arbitrary collection of cards with different face values. At the start of the game, the deck contains the same number of each cards of each face value; we call this number the 'multiplicity'. For example, a standard deck of 52 cards would have face values $[1, 2, \ldots, 13]$ and multiplicity 4. You could also have a deck with face values $[1, 5, 20]$; if we used multiplicity 10 in this case, there would be 30 cards in total (10 each of 1s, 5s, and 20s). The deck is shuffled, meaning that each permutation of the cards is equally likely.
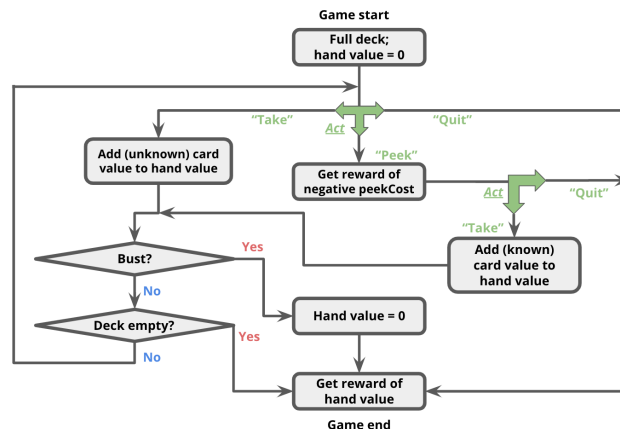
The game occurs in a sequence of rounds. Each round, the player either (i) takes the next card from the top of the deck (costing nothing), (ii) peeks at the top card (costing `peekCost`, in which case the next round, that card will be drawn), or (iii) quits the game. (Note: it is not possible to peek 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 face values of the cards in her hand.
- The player takes a card and "goes bust". This means that the sum of the face values of the cards in her hand is strictly greater than the threshold specified at the start of the game. If this happens, 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. *Make sure that if you take the last card and go bust, then the reward becomes 0 not the sum of values of cards.*

In this problem, your state s will be represented as a 3-element tuple:

`(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 of which has equal probability of $1/3$) are:

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

  She will receive a reward of 0 for reaching any of these states. (Remember, even though she now has a card in her hand for which she may receive a reward at the end of the game, the reward is not actually granted until the game ends.)

- If she 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 (immediate) reward `-peekCost` for reaching any of these states. Things to remember about the states after a peek action:

- From `(0, 0, (1, 1, 1))`, taking a card will lead to the state `(1, None, (0, 1, 1))` deterministically.
- The second element of the state tuple is not the face value of the card that will be drawn next, but the index into the deck (the third element of the state tuple) of the card that will be drawn next. In other words, the second element will always be between 0 and `len(deckCardCounts)-1`, inclusive.

- If she quits, then the resulting state will be `(0, None, None)`. (Remember that setting the deck to `None` signifies the end of the game.)

As another example, let's say the player's current state is `(3, None, (1, 1, 0))`, and the threshold remains 4.

- 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) [**8 points (Coding)**] Implement the game of Blackjack as an MDP by filling out the `succAndProbReward()` function of class `BlackjackMDP`.

(b) [**4 points (Coding)**] 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.

Hint: Before randomly assinging values, think of the case when you really want to peek instead of blindly taking a card.

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 its rules and strategy at the same time!

(a) [**8 points (Coding)**] 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 that 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 $\hat{Q}_{\mathrm{opt}}(s, a) = \lesssim \cdot \phi(s, a)$, where in code, $\lesssim$ is `self.weights`, ϕ is the `featureExtractor` function, and $\hat{Q}_{\mathrm{opt}}$ 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) [**5 points (Written)**] Now let's apply Q-learning to an MDP and see how well it performs in comparison with value iteration. First, call `simulate` using your Q-learning code and the `identityFeatureExtractor()` on the MDP `smallMDP` (defined for you in `submission.py`), with 30000 trials. How does the Q-learning policy compare with a policy learned by value iteration (i.e., for how many states do they produce a different action)? (Don't forget to set the explorationProb of your Q-learning algorithm to 0 after learning the policy.) Now run `simulate()` on `largeMDP`, again with 30000 trials. How does the policy learned in this case compare to the policy learned by value iteration? What went wrong?

(c) [**5 points (Coding)**] To address the problems explored in the previous exercise, let's incorporate some domain knowledge to improve generalization. This way, the algorithm can use what it has 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) [**5 points (Written)**] Sometimes, we might reasonably wonder how an optimal policy learned for one MDP might perform if applied to another MDP with similar structure but slightly different characteristics. For example, imagine that you created an MDP to choose an optimal strategy for playing "traditional" blackjack, with a standard card deck and a threshold of 21. You're living it up in Vegas every weekend, but the casinos get wise to your approach and decide to make a change to the game to disrupt your strategy: going forward, the threshold for the blackjack tables is 17 instead of 21. If you continued playing the modified game with your original policy, how well would you do? (This is just a hypothetical example; we won't look specifically at the blackjack game in this problem.)

To explore this scenario, let's take a brief look at how a policy learned using value iteration responds to a change in the rules of the MDP.

- First, run value iteration on the `originalMDP` (defined for you in `submission.py`) to compute an optimal policy for that MDP.
- Next, simulate your policy on `newThresholdMDP` (also defined for you in `submission.py`) by calling `simulate` with an instance of `FixedRLAlgorithm` that has been instantiated using the policy you computed with value iteration. What is the expected reward from this simulation?
  Hint: read the documentation (comments) for the `simulate` *function in util.py, and look specifically at the format of the function's return value.*
- Now try simulating Q-learning directly on `newThresholdMDP` with `blackjackFeatureExtractor` and the default exploration probability. What is your expected reward under the new Q-learning policy? Provide some explanation for how the rewards compare, and why they are different.