Assignment 4. Peeking Blackjack

Hwanjo Yu
CSED342 - Artificial Intelligence

Contact: TA Seongje Lee (lsj720@postech.ac.kr), Jaehyun Lee (jminy8@postech.ac.kr)

General Instructions

This (and every) assignment has a written part and a programming part.

You should write both types of answers in `submission.py` between

```
# BEGIN_YOUR_ANSWER
```

and

```
# END_YOUR_ANSWER
```

✎This icon means a written answer is expected. Some of these problems are multiple choice questions that impose negative scores if the answers are incorrect. So, don't write answers unless you are confident.

⌨This icon means you should write code. you can add other helper functions outside the answer 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`.

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.

Problem 1. Blackjack and Value Iteration

You will be creating a MDP to describe a modified version of Blackjack. (Before reading the description of the task, first check how `util.ValueIteration.solve` finds the optimal policy of a given MDP such as `util.NumberLineMDP`.)

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 card will be drawn in the next round), 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.

Problem 1a [5 points] ⌨

Your task is to implement the game of Blackjack as a MDP by filling out the `succAndProbReward()` function of class `BlackjackMDP`.

Problem 1b [5 points] ⌨

Now we'll implement a variation of value iteration specialized in acyclic MDPs. Your task it to implement `ValueIterationDP` which should use dynamic programming to compute the optimal values. The time complexity of the algorithm should be linear to the number of all possible transitions in an acyclic MDP. The implemented algorithm should be faster than the normal value iteration.

Problem 2: 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!

Problem 2a [5 points] ⌨

You will first implement a generic Q-learning algorithm `Qlearning`, 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 `Qlearning` will be run in a simulation of the MDP, and will alternately be asked for an action to perform in a given state (`Qlearning.getAction`), and then be informed of the result of that action (`Qlearning.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 `Qlearning.getAction` as a simple ϵ-greedy policy. Your job is to implement `Qlearning.incorporateFeedback()`, which should take an (s, a, r, s') tuple and update `self.weights` according to the standard Q-learning update.

Problem 2b [5 points] ⌨

Now, you'll implement SARSA, which can be considered as a variation of Q-learning. Your task is fill in `incorporateFeedback` of `SARSA`, which is same with `Qlearning` except the update equation.

Problem 2c [5 points] ⌨

Now, we'll incorporate domain knowledge to improve generalization of RL algorithms for `BlackjackMDP`. Your task is to implement `blackjackFeatureExtractor` as described in the code comments in `submission.py`. This way, the RL algorithm can use what it learned about some states to improve its prediction performance on other states. Using the feature extractor, you should be able to get pretty close to the optimum on some large instances of `BlackjackMDP`.