Assignment 4

Blackjack And Reinforcement Learning

Gary Geunbae Lee
CSED342 - Artificial Intelligence

Contact: TA Baikjin Jung (bjjung@postech.ac.kr)

General Instructions

All assignments consist of written parts and programming parts.

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 we recommend you not to 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`. Please rename the file `submission_{your student ID}.py` before handing it in.

Your code will go through two types of tests: **basic** and **hidden** tests. You can see that every step of a basic test is transparent (`grader.py`). It does not make your code handle a large input value or a tricky corner case. In contrast, you can see only the input values of hidden tests; the correct output values are not provided (`grader.py`). Also, these tests may make your code handle a large input value or a trick corner case. To run all the tests, type

```
python grader.py
```

It will tell you only whether you have passed the basic tests; for the hidden tests, the script will alert you if your code takes too long or crashes, but it will not tell you 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 encourage you to read and understand the test cases, create your own test cases, not just blindly run `grader.py`.

Problem 1. Blackjack And Markov Decision Process

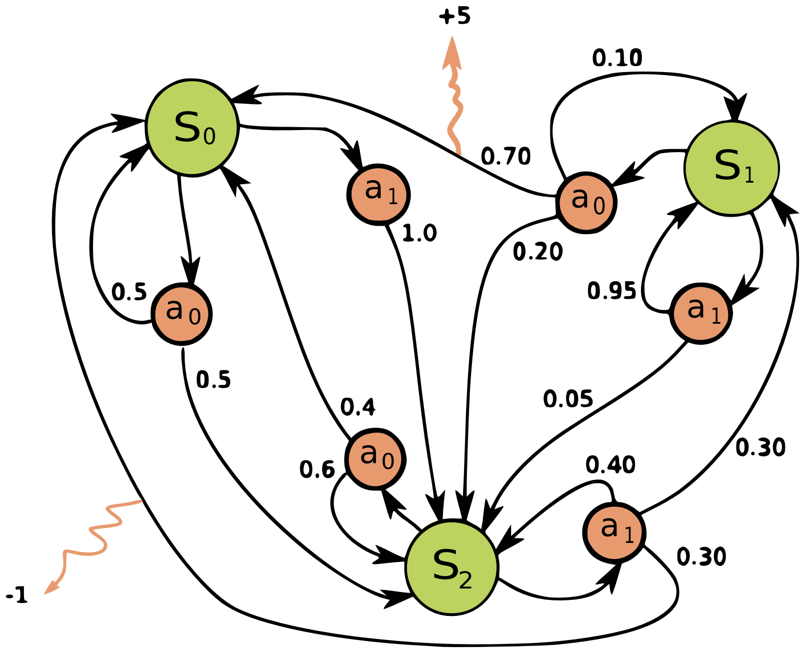

Figure 1: Example of a simple MDP with three states (green circles) and two actions (orange circles), with two rewards (orange arrows). (Waldoalvarez, 2017)

Your assignment is to implement an MDP that models a modified version of blackjack and extend it to a reinforcement learning model. Please read the detailed description below and follow the given instructions. Also, please look at how `ValueIteration.solve()` finds the optimal policy of a given MDP such as `NumberLineMDP` in `util.py`.

However, let's first look at the game itself. A deck of cards is supposed to contain a collection of cards with different values, each with a given multiplicity. For example, a *standard 52-card deck* have a collection $[1, 2, \cdots, 13]$ with a multiplicity 4 (clubs (♣), diamonds (♦), hearts (♥) and spades (♠)). However, you can also have a deck with cards $[1, 5, 20]$ with multiplicity 1. Also, the deck is supposed to be shuffled (each permutation of cards is equally likely).

A player is supposed to play several 'rounds'; for each round, he can either (i) draw a card from the top of the deck without cost, (ii) pay `peekCost` and secure himself a specific card that he can draw in the next round (we call it 'peek'), or (iii) quit the game. It is not possible to peek twice in a row; `succAndProbReward()` should return [] if the player peeks twice in a row.

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

1. the player quits,
2. the sum of the cards in the player's hand is strictly greater than the given threshold,
3. or the deck runs out of cards.

In the first and third cases, the reward that the player will obtain is equal to the sum of the cards in his hand while the player will obtain nothing (0) in the second case.

Now, let's design an MDP for this game. Every state s is represented as a triplet

`(totalCardValueInHand, nextCardIndexIfPeeked, deckCardCounts).`

Here is an example. Let's assume that we have a deck with values $[1, 2, 3]$ and multiplicity 1, and the threshold value is 4. Initially, the player has no cards, which means `totalCardValueInHand` is 0. Thus, $s = (0, \text{None}, (1, 1, 1))$. Now, he can choose one of the three actions: he can either draw a card, peek a card, or quit.

- If he draws a card, the set of all possible successor states

  $$S_{\text{next}} = \{(1, \text{None}, (0, 1, 1)), (2, \text{None}, (1, 0, 1)), (3, \text{None}, (1, 1, 0))\}.$$

  Here, each possible state has a possibility $\frac{1}{3}$.

- If he peeks a specific card, the set of all possible successor states

  $$S_{\text{next}} = \{(0, 0, (1, 1, 1)), (0, 1, (1, 1, 1)), (0, 2, (1, 1, 1))\}.$$

  Here, the player is going to reach the 'peeked' state deterministically. Also, `-peekCost` will be added to the reward value. Please note that you should draw the card in the next round; the designated card will not be drawn automatically.

- If he quits, then the resulting successor state will be $(0, \text{None}, \text{None})$ (`deckCardCounts` being None means that the game is over).

In the same environment, let's say this time that the player's current state is $(3, \text{None}, (1, 1, 0))$. He can either draw a card, peek a card, or quit.

- If he draws a card, the set of all possible successor states

  $$S_{\text{next}} = \{(3 + 1, \text{None}, (0, 1, 0)), (3 + 2, \text{None}, \text{None})\}.$$

  Here, each possible state has a possibility $\frac{1}{2}$. In the latter case, the game must end with a bust (exceeding the threshold). Including that case, whenever the game ends, `deckCardCounts` should be None.

- If he peeks a specific card, the set of all possible successor states

  $$S_{\text{next}} = \{(3, 0, (1, 1, 0)), (3, 1, (1, 1, 0))\}.$$

  `-peekCost` will be added to the reward value.

- If he quits, then the resulting state will be $(3, \text{None}, \text{None})$.

Problem 1a [10 points] ⌨

Implement this game by filling out the `succAndProbReward()` function of the `BlackjackMDP` class.

Problem 1b [5 points] ⌨

Implement `ValueIterationDP`, a variation of value iteration specialized in acyclic MDPs, 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. Blackjack And Reinforcement Learning

Suppose that we don't know the full dynamics of the game, which means estimating an optimal policy before actually playing the game is impossible. In such circumstances, reinforcement learning can allow you to play the game and learn the rules at the same time!

Problem 2a [5 points] ⌨

Implement a generic Q-learning algorithm `Qlearning`, which is a subclass of `RLAlgorithm`. As discussed in class, reinforcement learning algorithms are capable of executing a policy while simultaneously improving their policy.

Please Look into `simulate()` in `util.py` to see how `RLAlgorithm` is 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 $\hat{Q}_{opt}(s,a) = \mathbf{w} \cdot \phi(s,a)$, where $\mathbf{w}$ corresponds to `self.weights`, ϕ `self.featureExtractor()`, and $\hat{Q}_{opt}$ `self.getQ()`.

Because `Qlearning.getAction()` is already implemented as a simple ϵ-greedy policy, your job is to implement `Qlearning.incorporateFeedback()`, which should take a tuple $(s, a, r, s\prime)$ and update `self.weights` by using the standard Q-learning update.

Problem 2b [5 points] ⌨

Implement SARSA, which is a variation of Q-learning. Fill in `incorporateFeedback()` of `SARSA`, which is the same as in `Qlearning` except the update equation.

Problem 2c [5 points] ⌨

Let's incorporate domain knowledge to improve the generalization of the RL algorithm. Implement `blackjackFeatureExtractor` as described in the code comments in `submission.py`. This way, the RL algorithm will use what it has learned about certain states to improve its prediction performance on other states. Using the feature extractor, you should be able to get pretty close to the optimal performance, even for some large instances of `BlackjackMDP`

References

Waldoalvarez. (2017). Markov Decision Process [Diagram]. *Wikimedia*. https://commons.wikimedia.org/w/index.php?curid=59364518