MDP Problem: Peeking Blackjack

Shamelessly adapted from Stanford CS-221 Class

November 29, 2023

Abstract

Look into analyzing the BlackJack game as an MDP

1 Intro

Now that we have gotten a bit of practce 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. More specifically, after reading through the description of the state representation and actions of our Blackjack game below, you will implement the transition and reward function of the Blackjack MDP inside the BlackJackMDP calsss.

2 Our Version of BlackJack

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, ... , 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. In each round, the player has three actions available to her:

- a_{take} - Take the next card from the top of the deck.
- a_{peek} - Peek at the next card on the top of the deck.
- a_{quit}- Stop taking any more cards.

Here is a graphical view of it:

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

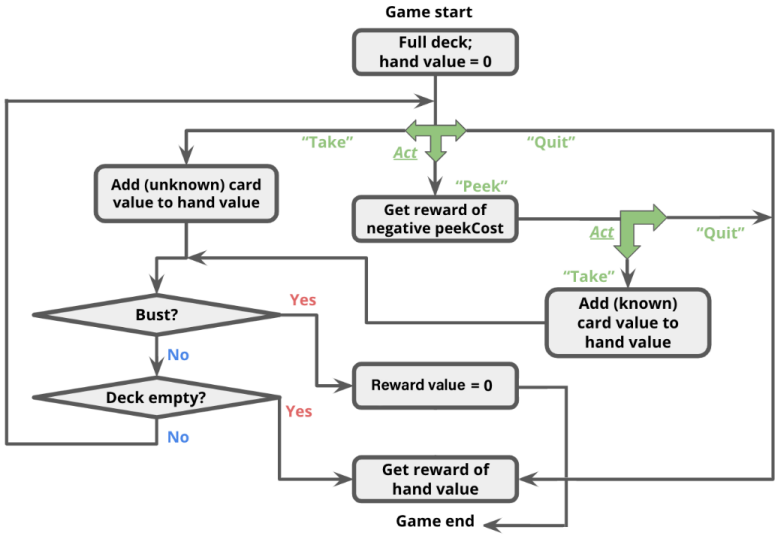

Figure 1: Flow of BlackJack game

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

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

- For a_{take}, the three possible successor states (each with equal probability of 1/3) are:

  (1, `None`, (1, 2, 2))

  (2, `None`, (2, 1, 2))

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

  Three successor states have equal probabilities because each face value had the same amount of cards in the deck. In other words, a random card that is available in the deck is drawn and its corresponding count in the deck is then decremented. Remember your implementation of `transitionProbs()` will expect you return all three of the successor states shown above.

  Note that $R(s, a_{take}, s') = 0, \forall s, s' \in S$. Even though the agent 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 (see termination conditions below).

- For a_{peek} , the three possible successor states are:

(0, 0, (2, 2, 2))

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

(0, 2, (2, 2, 2))

Note that it is not possible to peek twice in a row; if the player peeks twice in a row, then `transitionProbs()` should return []. Additionally, $R(s, a_{peek}, s') = -peekCost, \forall s, s' \in S$ That is, the agent will receive an immediate reward of -peekCost for reaching any of these states.

Things to remember about the states after taking a_{peek} :

- From (0, 0, (2, 2, 2)), taking a card will lead to the state (1, `None`, (1, 2, 2)) deterministically (that is, with probability 1.0).
- 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.

- For a_{quit}, the resulting state will be (0, `None`, `None`). (Remember that setting the deck to `None`signifies the end of the game.)

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

1. The player chooses a_{quit} , in which case her reward is the sum of the face values of the cards in her hand.
2. The player chooses a_{take} 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.
3. The deck runs out of cards, in which case it is as if she selects a_{quit} , 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.

As another example with our deck of [1, 2, 3] and multiplicity 1, let's say the player's current state is (3, `None`, (1, 1, 0)), and the threshold remains 4.

- For a_{quit}, the successor state will be (3, `None`, `None`).
- For a_{take}, the successor states are (3 + 1, `None`, (0, 1, 0)) or (3 + 2, `None`, `None`). Each has a probability of 1/2 since 2 cards remain in the deck. 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.

3 What is requested

You may use the mdp framework, that includes the key algorithms for **ValueIteration** and **PolicyFromValue** provided in the github at `https://github.com/juanclaudioagui/FIA23-24-Alumnos/tree/main/MDP`

1. Implement the game of Blackjack as an MDP by filling out the methods from the mdp class.

   If some of your test cases for 3a are failing, here are some tricky questions to consider:

   - What happens if you "peek" twice in a row?
   - Can you peek if you have no cards left?
   - If you peeked before, in the next immediate round, which card are you guaranteed to take?
   - If the card value exceeds the threshold and you go bust, how do you indicate that you went bust in your successor state?
   - Does the probability of taking a card differ depending on how many of such cards there are in the deck?

2. Write a simple code play generate and play the PeekBlackJack game manually. That is, start on `(0,None,(n,..n))` where n is the multiplicity and tell it which action among the TAKE, PEEK, or QUIT you want to take.

   Play it several times and see how good you are, even with the help of the card counts being taken care for you by the game[1]. Toma nota de tus marcas finales.

3. Identify and generate all possible states in the game. Essentially is about all combinations of cards, within the given multiplicity and under the given threshold. Remember that, for every basic state, we will have the same version, as the result of the **QUIT** action, and some other states that come as a result of invoking the **PEEK**

4. **Optionally** write a `Value Iteration` algorithm, following the example provided in Russel & Norvig [2] with no time degradation, that is, with $\lambda = 1$. (Alternatively, you can use the code provided in the github repository)

5. **Optionally** write a `PolicyFromValue` algorithm, which simply derives the optimal policy $\wp(si) \rightarrow a_i$ to take at any of the possible states. Lístala, dibújala, entiéndela !

6. **Optionally** Puedes ahora utilizar esta política óptima generada como una

[1] A mucha gente las han echado de los casinos por contar cartas...
[2] see Book in page 563

ayuda, que te recomiende una acción para cada estado, dentro de un juego manual.

7. **Optionally** Implementa un modo automático de juego, cuyo resultado, para cada deck aleatorio concreto que generes de forma aleatoria, juege con la estrategia óptima y obtenga un resultado. Juégalo muchas veces (100,1000,...) y haz una estadística de los reslutados (Veces que te estrellas, valor medio, varianza, etc..) y compara con tu marca manual que hiciste antes.