ECE 473: Introduction to Artificial Intelligence

Assignment: Peeking Blackjack

ECE473

Submission Instructions

- Only submit `hw5_submission.py` and do not change the filename. Our autograder will find "`hw5_submission.py`" to test your assignment. If you change the filename, autograder will crash and you will receive 0 points.
- You are responsible for checking your program runs properly and run on any system.
- Submit early and often. You can submit as many times as you'd like until the deadline: we will only grade the last submission.
- We will not be accepting any assignments submitted via email.
- You should modify the code in `hw5_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 `hw5_submission.py`.
- Your code will be evaluated on two types of test cases, basic and hidden, which you can see in `hw5_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 `hw5_grader.py`, but the correct outputs are not. To run the tests, you will need to have `graderUtil.py` in the same directory as your code and `hw5_grader.py`. Then, you can run all the tests by typing `python hw5_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 hw5_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 `hw5_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: Transforming MDPs

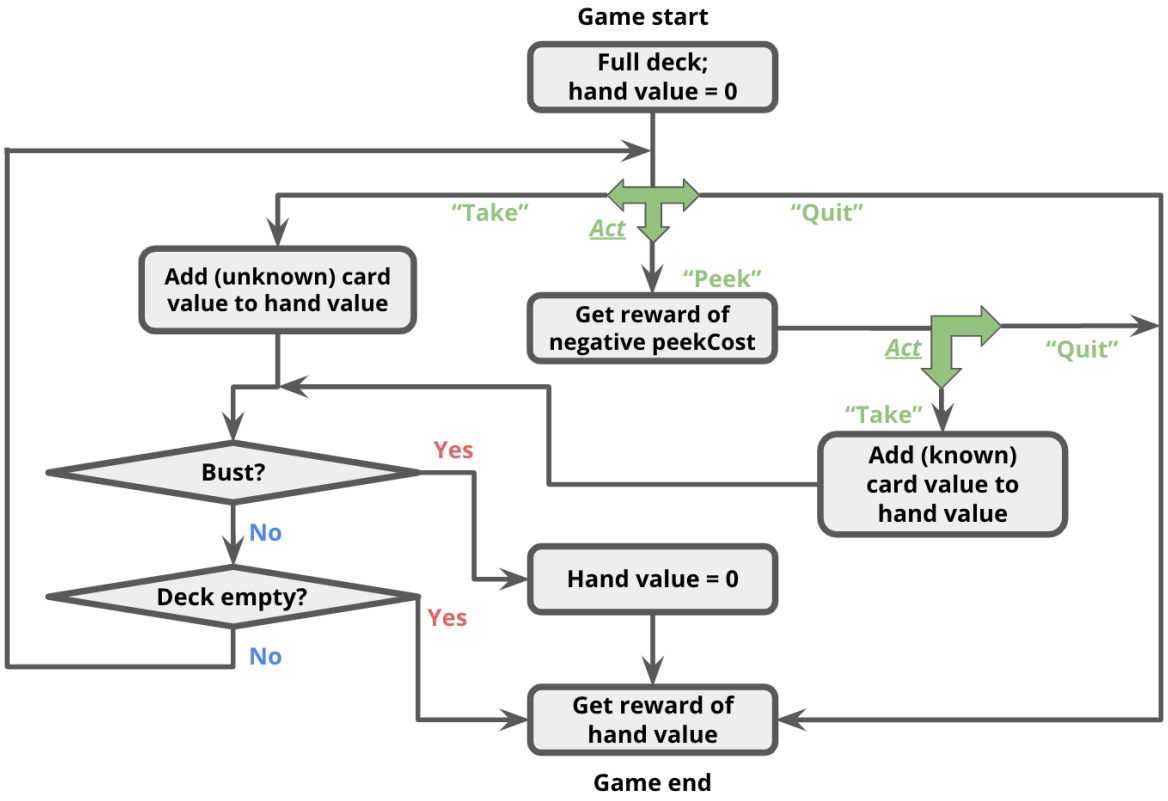
Equipped with an understanding of a basic algorithm for computing optimal value functions in MDPs, let's gain intuition about the dynamics of MDPs which either carry some special structure, or are defined with respect to a different MDP. If we add noise to the transitions of an MDP, does the optimal value always get worse? Specifically, consider an MDP with reward function $R(s, a, s')$, states S, 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 $V_1(s_{start}) \geq V_2(s_{start})$? If so, put `return None` for each of the code blocks. Otherwise, construct a counterexample by filling out `CounterexampleMDP` in `hw5_submission.py`.

Problem 2: 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. 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 `succAndProbReward()`.



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. 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_{stop} - Stop taking any more 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))`

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

  `(1, None, (0, 1, 1))`

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

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

  In words, a random card that is available in the deck is drawn and its corresponding count in the deck is then decremented. Remember that `succAndProbReward()` will expect you return all three of the successor states shown above. 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, (1, 1, 1))`

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

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

  Note that it is not possible to peek twice in a row; if the player peeks twice in a row, then `succAndProbReward()` should return `[]`. Additionally, $\mathcal{R}(s, a_{\text{peek}}, s') = -\text{peekCost}, \forall s, s' \in \mathcal{S}$. 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, (1, 1, 1))`, taking a card will lead to the state `(1, None, (0, 1, 1))` 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_{stop}, 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:

- The player chooses a_{stop}, in which case her reward is the sum of the face values of the cards in her hand.
- 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.
- The deck runs out of cards, in which case it is as if she selects a_{stop}, 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_{stop}, 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)`. 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. Implement the game of Blackjack as an MDP by filling out the `succAndProbReward()` function of class `BlackjackMDP`.

b. 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.]

Problem 3: 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 & strategy at the same time!

a. 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 $\hat{Q}^\star(s,a) = \mathbb{w} \cdot \phi(s,a))$, where in code, $\mathbb{w}$ is `self.weights`, ϕ is the `featureExtractor` function, and $\hat{Q}^\star$ 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. 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 `hw5_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?

   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`.

c. 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. For all subsequent parts, make sure to use 30,000 trials.

   - First, run value iteration on the `originalMDP` (defined for you in `hw5_submission.py`) to compute an optimal policy for that MDP.
   - Next, simulate your policy on `newThresholdMDP` (also defined for you in `hw5_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.
- Provide the answers to the above questions in the comment at the very end of the `hw5_submission.py`.

Acknowledgement

Stanford CS221: Artificial Intelligence: Principles and Techniques