Report on Project 3: Black Jack

Ao Wang, 15300240004

May 21, 2018

1 Overview

This project is about solving Black Jack using MDPs. This pdf contains the answers of the required questions.

2 Problem 1

2.1 Problem 1a

As required by the instructions, the state -2 and 2 are the exits, so I manually set the $V_{opt}(s)$ of them to a fixed number, which is 0. Then the results are as follows:

Table 2.1: The result of 1st value iteration (after 0 iterations).

State s	-2	-1	0	1	2
$V_{opt}(s)$	0.0	**11.0**	**-5.0**	**25.0**	0.0

Table 2.2: The result of 2nd value iteration (after 1 iterations).

State s	-2	-1	0	1	2
$V_{opt}(s)$	0.0	**10.0**	**10.2**	**21.5**	0.0

Table 2.3: The result of 3rd value iteration (after 2 iterations).

State s	-2	-1	0	1	2
$V_{opt}(s)$	0.0	**13.04**	**8.45**	**32.14**	0.0

2.2 Problem 1b

After the iteration is converged, the result of π_{opt} is as follows:

Table 2.4: The result of the optimal actions after being converged.

State s	-2	-1	0	1	2
$\pi_{opt}(s)$	-	**-1**	**+1**	**+1**	-

The result is pretty clear because the agent tries to get to the exit as soon as possible as the reward of each step is negative and both of the exits have comparatively high positive rewards. But at state 0, agent will have higher expectation by moving +1, although it has a high possibility of moving -1. The details are in the `submission.py`.

3 Problem 2

3.1 Problem 2a

According to the instruction, the transition (probability) has been changed with noise, with a probability to randomly change to a possible reachable state. I can find a counter example that $V_1(s_{start}) \leq V_2(s_{start})$.

I construct a MDP with three states: -1, 0 and 1. -1 is the exit state with a reward of -20, but 0 and 1 have positive rewards of 10 and 20. In the original MDP, state 0 and state 1 will transfer to state -1 with a probability of 1.0, end with a reward of -20. After adding noise, the state 0 and 1 have possibility to transfer to state 0 and 1 with high rewards, definitely making the $V(start)$ higher than original ones.

3.2 Problem 2b

Since this MDP is acyclic, I can use *Dynamic Programming* in a iteration style to solve this problem with the constraints of traversing each (s, a, s') triple only once.

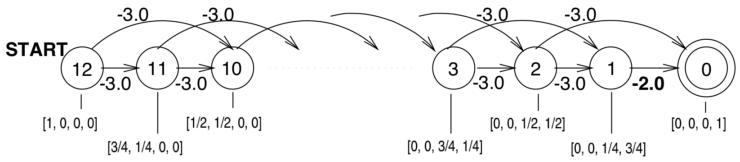

Figure 3.1: An example of acyclic MDP.

The acyclic MDP may look like Figure 3.1. For this case, I give the dynamic programming algorithm to solve this problem. Note that the triple (s, a, s') can be regarded as the edge in the acyclic graph.

Algorithm 1 Dynamic Programming for Acyclic MDP

Input: s, s': state; a: action; γ: discount factor; $A(s)$: the feasible action set of state s; $V(s)$: the optimal value of state s; $R(s)$: the reward of state s; $T(s, a, s')$: the transition from state s to s' by taking action a
Output: V: the optimal values of all states

$V(s_{start}) = R(s_{start})$
unexplored = $List(S)$ `//add all states to the unexplored list`
$s = s_{start}$
while unexplored.$isNotEmpty$ **do**
    unexplored.pop(s) `//pop the current state` s `to avoid repetition`
    children_list = $Successor(s)$ `//all the reachable state from state` s
    for $s' \in children\_list$ **do** `//select next state`
        if $\forall s'' \in Ancestor(s')$, $V(s'')$ is settled **then**
            $s = s'$ `//move to next calculable state, with all ancestors settled`
            Break
        end if
    end for
    value_list = [] `//carry out DP`
    for $s' \in Ancestor(s)$ **do** `//value iteration`
        value_list.$append((a, R(s') + \gamma T(s', a, s)V(s)))$ `//the triple is used only once`
    end for
    $a_{opt}(s), V_{opt}(s) = max$(value_list)
end while

The basic rationale of this DP algorithm is pretty simple by calculating the optimal value of each state one by one. Note that there is a key procedure to ensure all of the ancestor states of the feasible state are settled in that the DP needs all of its ancestors' optimal values. Take the example of Figure 3.1. After settling the start state 12, we can go to 11 or 10. But one of the ancestor of state 10 is 11 and the value of state 11 is not settled yet, so I have to choose state 11 as the next state to carry out DP. I ensure every (s', a, s) triple is used only once by using a list to record every unexplored state. During the explore of each state s, the

ancestor state s' is explored only once. As a consequence, the state s and s' are used together only once, thus the (s', a, s) triple is used only once.

3.3 Problem 2c

As we can see from the Bellman equation:

$$V_{opt}(s) := R(s) + \gamma \max_{a \in A(s)} \sum_{s'} T(s, a, s') V_{opt}(s') \tag{3.1}$$

The discount factor γ multiplies the product of $T(s, a, s')$ and $V_{opt}(s')$, so after adding a new state o, we can change the $T(s, a, s')$ and $R(s, a, s')$ as follows:

$$\begin{cases} T'(s, a, s') &= \gamma T(s, a, s') \ if \ s' \neq o \\ T'(s, a, o) &= 1 - \gamma \\ Reward'(s, a, s') &= Reward(s, a, s') \ if \ s' \neq o \\ Reward'(s, a, o) &= \sum_{s' \in States} T(s, a, s') Reward(s, a, s') \end{cases}$$

The basic idea is that if we add discount factor $\gamma < 1$ in the value iteration based on original MDP solver, the left losing part should be compensated by the virtual state o.

4 Problem 3

4.1 Problem 3a

Simply follow the instructions to build the MDP. Notice that the probability is actually the probability of this number being taken, which is $\frac{\#\ of\ this\ number}{\#\ of\ all\ numbers}$.

4.2 Problem 3b

It's rather easy to design a deck to make people peek a lot. The basic idea is that if it's very easy to exceed the threshold to get 0 reward, the number of peek optimal action will increase to see if it's going to exceed. To accomplish this goal, it only need to add a rather big number, say 15, to the deck to make it easy to exceed the threshold 20.

5 Problem 4

5.1 Problem 4a

To implement a standard Q-learning process.

5.2 Problem 4b

The test code is in `submission.py` in function `test1`. Run `submission.py` directly to see the results.

Compared the result policies from Q-learning with 30,000 iterations and from value iteration, the conclusions are as follows. In `smallMDP`, the results from Q-learning simulation is nearly the same with value iteration, with only 0/1/2/3 wrong results out of 27 states. After many tries, the number of correct optimal policy is pretty high, although it can not guarantee an absolute correctness in an experiment. However, in `largeMDP`, the error rate is very high, about 800~900/2745, even with an iteration of 30,000 rounds.

The reason may be that the learning rate is **fixedly** related to the number of iteration (as we can see from the function `getStepSize`). As a consequence, when it comes to a large MDP that needs many iterations to converge, the learning rate may quickly reduce to 0 (not taking information from new iterations any more) before the Q-learning is converged.

5.3 Problem 4c

Just follow the instructions to extract features from current state.

After the completion of function `blackjackFeatureExtractor`, I implement the test code in `test2`. The error rate reduces indeed.

5.4 Problem 4d