Programming Assignment 3: Reinforcement learning

Assigned: October 16
Due date: November 18

1 General idea

In this assignment, you will write a program that uses reinforcement learning to learn the correct policy in a Markov Decision Process.

The input to the program is a Markov Decision Process. Rather than solve this, however, the program will learn what to do through experimentation. It will play many rounds. In each round it starts in a random state. It then goes through the process: at each state it chooses an action, and then the stochastic process puts it in a new state. When it reaches a terminal state, that round is over. It now records, for each state and choice of action that it went through, how well that choice of action worked out for it.

Initially it chooses the action at random, but as it plays more and more rounds and gets more and more experience, it increasingly chooses the actions that in the past have paid off for it.

Conceptually, there is an agent and a world model. The world model randomly chooses a starting state and executes the Markov decision process each time the agent chooses an action. The agent chooses actions and gradually learns what actions are best. You are not required to implement these as separate modules, though you might wish to.

2 Markov decision process

There are n non-terminal states, numbered 0 to $n-1$ and t terminal states, numbered n to $n+t-1$. In each state, there is at least one action, numbered consecutively from 0.

The transition probabilities for any given action in any given situation are given in the program input.

In each round, the agent starts at a random starting state. The agent repeatedly chooses an action according to a formula discussed in section 6, and the world model moves them to a new state, until the agent reaches a terminal state, when the round ends. When the round ends, the agent updates the information about the states traversed and the actions chosen as discussed below in section 4.

3 Input

The input is from a text file. The name of the text file should be given on the command line: E.g. `java prog3 input.txt` or `python3 prog3 input.txt`.

The first line consists of five parameters:

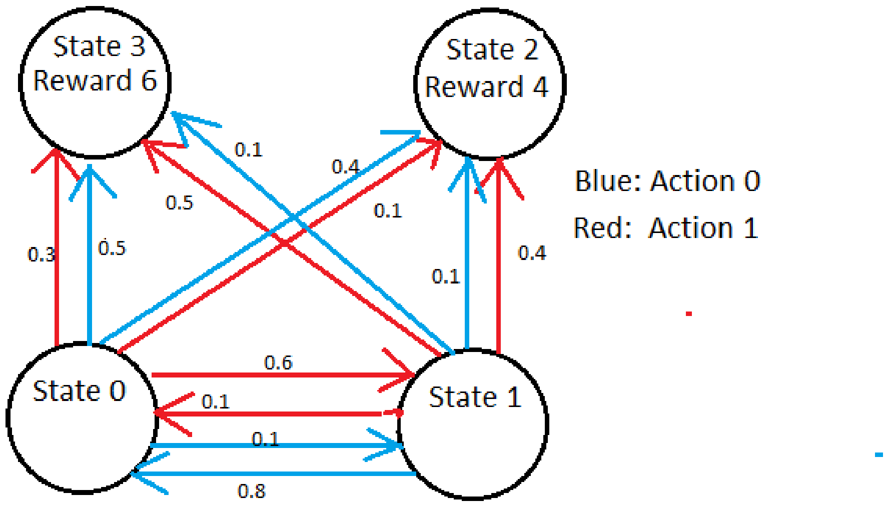
1. The number of non-terminal states.
2. The number of terminal states.
3. The number of rounds to play,

4. The frequency with which to print output (section 5).
5. The parameter M (an integer) controlling the "explore/exploit" trade-off discussed in section 6

The second line specifies the rewards in the terminal states, alternating state number with its reward (an integer).

The remaining line specify the transition probabilities for actions in non-terminal states. Each line begins with the number of a state, colon, the number of a action followed by alternating a number of a state number and the probability of transitioning to that state.

3.1 Example Input



For instance, here is the input for the simple process above (similar to the one in problem set 5, but with numbers on the state, and with different rewards):

```
2 2 3 2 10
2 4 3 6
0:0 1 0.1 2 0.4 3 0.5
0:1 1 0.6 2 0.1 3 0.3
1:0 0 0.8 2 0.1 3 0.1
1:1 0 0.1 2 0.4 3 0.5
```

You may assume that the input is correctly formatted and describes a valid MDP. You may also assume that from any starting state, any policy has probability 1 of eventually reaching a terminal state, so you don't have to worry about a round going into an infinite loop.

Self-loops are possible; that is, there may be an action that has some chance of leaving the agent in the same state.

4 Learning

Set up a data structure that records for each non-terminal state S and action A:

- `Count[S,A]` The number of rounds so far that the agent has been in state S and chosen A.
- `Total[S,A]` The total reward of those rounds.

During each round, keep a data structure that tracks the states that have been encountered and the actions taken. (Do not count duplicates.) At the end of the round, for each state-action `S,A` encountered, increment `Count[S,A]` by 1 and increase `Total[S,A]` by the reward for the round.

For instance, suppose that we play three rounds.

The first round is the sequence `0:1, 1:0, 3` (that is, the agent starts at state 0; it take action 1; that takes it to 1; it takes action 0; that takes it to the terminal state 3 where it gets a reward of 6.) `Count[0,1]` and `Count[1,0]` will be set to 1. `Total[0,1]` and `Total[1,0]` will be set to 6.

The second round is the sequence `1:1, 0:1, 1:1, 2` `Count[1,1]` will be incremented by 1 (not 2, even though it appears twice in the round. `Count[0,1]` will be incremented by 1. `Total[0,1]` and `Total[1,1]` will be increased by 4, the payoff for the round.

The third round is the sequence `1:0, 0:1, 1:0, 0:0, 3` `Count[1,0]`, Count[0,1], and Count[0,0] will be incremented by 1. `Total[1,0]`, `Total[0,1]`, and `Total[0,0]` will be increased by 6.

So at the end the three rounds,
`Count[0,0]=1; Count[0,1]=3; Count[1,0]=2; Count[1,1]=1.`
`Total[0,0]=6; Total[0,1]=16; Total[1,0]=12; Total[1,1]=4.`

5 Output

The amount of output is determined by the frequency parameter v in the input. If $v = 0$, then the output should be printed only after all the rounds are complete; otherwise, output should be printed after every v rounds.

When output is printed, it should show:
The current values of the `Count` and `Total` matrices. The best move in each state S, as determined by the largest value of `Total[S,A] / Count[S,A]`. If `Count[S,A]` is 0 for any action `A`, output `U`, for unknown.

5.1 Example Output

In the example above, the output would be:

```
After 2 rounds
Count:
[0,0]=0. [0,1]=2.
[1,0]=1. [1,1]=1.

Total:
[0,0]=0. [0,1]=10.
```

```
[1,0]=6. [1,1]=4.

Best action: 0:U.  1:0.

After 3 rounds
Count:
[0,0]=1. [0,1]=3.
[1,0]=2. [1,1]=1.

Total:
[0,0]=6. [0,1]=16.
[1,0]=12. [1,1]=4.

Best action: 0:0. 1:0.
```

6 Choosing the Action

When one is combining actions with learning in an unknown environment, there is generally an "explore/exploit" trade-off to consider. That is, how much effort should you spend exploring all the possibilities, versus exploiting actions that have been found to be successful. Generally, when you are starting in a new environment, you want to favor exploring; as you get to know the environment, you increasingly favor exploiting it. For instance, if you move to a new city and enjoy eating out, then at first, you try lots of possibilities; after you've been there for a while, you increasingly go to old favorites.

In this assignment, the way in which the action is chosen in a state goes from exploring (choosing an action at random) to exploiting (choosing the best action) gradually, at a speed controlled by the hyperparameter M; the larger M, the longer it takes to change from one regime to the other.

Your program should choose the action according to the following algorithm:

```
chooseAction(State s; int[][] count; int[][] total; int M) {        % M is the hyperparameter in the input;
    n = number of actions feasible in s;
    **if** (there are any actions u such that count[a,u]==0)
          **return** u;                                             % choose an untried action arbitrarily;
    **for** (i=0 to n-1) avg[i] = total[s,i]/count[s,i];           % average reward
    bottom = the smallest reward of any terminal node;
    top = the largest reward of any terminal node;
    **for** (i=0 to n-1) savg(i) = 0.25+0.75*(avg[i]-bottom)/(top-bottom)
                                                                    % average scaled to range [0.25,1];
    c = $\sum_{i=0}^{n-1}$ count[s,i];
    **for** (i=0 to n-1) up[i] = savg[i]**(c/M)                    % unnormalized probability
    norm = $\sum_{i=0}^{n-1}$ up[i];
    **for** (i=0 to n-1) p[i] = up[i]/norm;
    return (randomly choose action i with probability p[i]);
}
```

So, for instance suppose that the program has gone through the three rounds in the above example, so that
`Count[0,0]=1; Count[0,1]=3; Count[1,0]=2; Count[1,1]=1.`
`Total[0,0]=6; Total[0,1]=16; Total[1,0]=12; Total[1,1]=4.`

Let M=10. You are now playing a fourth round and the current state is 0. Then:
`avg=[6,5.3333].  top=6.  bottom=4.  savg=[1,0.75].`
`c=4.  up=`$[1^{0.4}, 0.75^{0.4}]$` = [1.0, 0.89].  p = [0.53, 0.47].`

So there is a 53% chance of choosing action 0 and a 47% chance of choosing action 1.

The details of the above are not particularly significant, or especially well adapted to this problem; in fact, I pulled them out of a hat. But the key points are this:

- In the early rounds, `c` will be much less than `M` so the exponent `c/M` will be much less than 1. When we raise the numbers in `savg`, which are all values between 0.25 and 1, to the power `c/M`, they will all be close to 1. So you are choosing nearly randomly.
- In the late rounds, `c` will be much greater than `M` so the exponent `c/M` will be much greater than 1. If i is the best move and j is the value of the second-best move, then up[i]/up[j] will be (savg[ii]/savg[j])**(c/M), which will be very large unless j is nearly as good as i. So with high probability, we choose the best move, or some move that is almost as good as the best move (as compared to the range in reward values at the terminal nodes).

Note. If `c` is much greater than `M`, then you run into the danger of underflow. If you want to be really fancy, you can program up an arithmetic exception handler, and change any value that is in underflow to 0 in `up`. Or you can write code to catch these some other way. But you are not required to bother with this; you can assume that the input parameters will be chosen so that it will not run into underflow.

7 Choosing randomly from a distribution

If you are coding in Python, you can use the method `random.choices()` described here: `https://docs.python.org/3/library/random.html`

In other languages: If you are given a distribution $p_0 \dots p_k$, then the way to sample from the numbers $0 \dots k$ with that distribution is to execute the following code: (The function rand() here generates a random floating point number uniformly distributed between 0 and 1. Any half-decent mathematical library will supply this with some name.)

```
$u_0 = p_0$
for ($i = 1 \dots k$) $u_i = u_{i-1} + p_i$.
$x$ = rand(); % Uniformly distributed between 0 and 1.
for ($i = 0 \dots k-1$)
        if $x < u_i$ return $i$;
return $k$; % You don't want to count on $u_k$ being exactly 1, because of round-off error.
```

For example, suppose that $p_0 = 1/8, p_2 = 3/8, p_3 = 1/6, p_4 = 1/3$.
Then $u_0 = p_0 = 1/8$.
$u_1 = p_0 + p_1 = 1/2$.
$u_2 = p_0 + p_1 + p_2 = 2/3$.
$u_3 = p_0 + p_1 + p_2 + p_3 = 1$.
So now if you pick the value x uniformly between 0 and 1, the probability that x is between 0 and u_0 is p_0; the probability that x is between u_0 and u_1 is p_1; and so forth.

8 Checking for correctness

I'll post some more inputs and outputs later.

Since this calls a random number generator, you will get somewhat different outputs from mine; in fact, you will get different results each time you run it. (It's a good idea, in debugging, to set the random number generator with a seed, so that, if you run into a bug, you can replicate the behavior. Different languages have different ways to do this.)

In debugging, run it with very few rounds and a small model, and do traces to make sure that it is doing the right thing.

Once it's working on the small model, test it with some reasonable number of rounds — say 100. Try different values of M — 10, 20, 50, 100 —, and set the output parameter to be some fraction of M. What you should see is:

- In the very early rounds, the `Count` values on the different actions for any given state will be small and completely random.
- After some more rounds, the `Count` values will fairly equal, with some small preference to the best action.
- When the number of rounds becomes bigger than M, the best actions will have significantly larger counts than inferior actions.
- When the number of rounds becomes much bigger than M, then the count on inferior actions will largely stop increasing.
- If you compare the computed expected value for each state `[Total[S,A]/Count[S,A]` at the end, that should be approximately equal to the correct value for the best move `A`. It is likely to be noticeably more accurate for the states that are close to the terminal states than for the states that are far from the terminal states.