Some Advice on the Reinforcement Learning Assignment

TL;DR version

1. You can expect significant discrepancies between your solution and my posted solutions.
2. Write your code in modules, and test bottom up. (see p. 4).

Exact solution

It is in fact feasible to compute the exact solution to this class of games: what is the best move, and what is the probability of winning, assuming that both players make the best move.

For instance, with the game `NDice=2, NSides=2, LTarget=4, UTarget=4` (that is, you are flipping either 1 or two coins; heads counts as 2 points, tails counts as 1 point; the goal is to get 4 points exactly), the exact result is as follows

Play

2	2	2	2
2	2	2	2
1	1	1	1
1	1	1	1

Prob:

0.5703	0.6094	0.4688	0.6250
0.5938	0.6250	0.5625	0.6250
0.6875	0.7500	0.6250	0.7500
0.5000	0.5000	0.5000	0.5000

These results are computed backward, from high scores to low. If the score is 3 to X, and you roll 1 die, then there is a 1/2 chance you will get 1, in which case you win, and a 1/2 chance that you will get 2, in which case you lose. If you roll 2 dice, then your roll will total at least 2, so you will necessarily lose. So you do better to roll 1 die, and your probability of winning is 0.5.

Now, suppose the score is 2 to 3. If you roll 1 die then the chance is 0.5 that you will roll 1. If you do, then the score will be 3 to 3, and we've already calculated that your opponent has a 0.5 chance of winning from that position. There is also a 0.5 chance that you will roll 2, in which case you win. So your total chance of winning, rolling 1 die, is 0.75. On the other hand, if you roll 2 dice, then there is a 1/4 chance of rolling a total of 2 and winning and 3/4 chance of rolling 3 or 4, and losing. So you do better to roll 1 die, and if you do, you have a 0.75 chance of winning.

And you can continue this kind of calculation back to 0-0. It's a nice little program, but it's not what the assignment is. It is convenient for computing correct "gold standard" answers that you can compare against.

I'll post a couple more exact answers on the web site.

Accuracy you can expect from your program

However, you can't expect the answers from your reinforcement learning program to be very close to the exact results. Here were the results on three runs of my RL program for the above game, with the hyperparameter `M = 100` and with `NGames = 100000`. (Since the programs make random choices, different runs give somewhat different results.) runs gave the same "Play" array, though not the one given in the correct solution:

First run:

Play

2	2	2	2
0	2	2	2
0	1	1	1
0	1	1	1

Prob:

0.5600	0.5455	0.4812	0.6402
0	0.5682	0.6148	0.6331
0	0.7355	0.6196	0.7624
0	0.5044	0.4987	0.4942

Second run:

Play:

2	2	2	2
0	2	2	1
0	1	1	1
0	1	1	1

Prob:

0.5625	0.5466	0.4743	0.6388
0	0.5385	0.5764	0.6538
0	0.6950	0.6370	0.7607
0	0.5890	0.5042	0.4973

Third run:
Play:

2	2	2	2
0	2	2	2
0	1	1	1
0	1	1	1

0.5637	0.5524	0.4797	0.6363
0	0.6042	0.5813	0.6651
0	0.6726	0.6288	0.7583
0	0.4882	0.5008	0.5001

What's going on here? There are a couple of issues, discussed in the assignment. The first issue is that some scores are completely impossible to attain, however the two players play. In this came, those are the scores $\langle 1, 0\rangle$, $\langle 2, 0\rangle$, and $\langle 3, 0\rangle$; if you (the current player) have a non-zero score, then this must be at least your second turn, and if so, your opponent must have reached a score of at least 1 on their first turn. Since these scores have never been reached, that means that the values of `WinCount[I,0,K]` and `LoseCount(I,0,K)` for I=1, 2, or 3 and for K=1,2 are all 0. We therefore output a play of 0 and a probability of 0 for these, to indicate that we have no information.

The second issue is that some scores can't be reached if both players make the correct moves. In this game, both players should play 2 dice on their first roll. This is intuitively obvious (if they roll 2 dice, they have a 1/4 chance of winning outright, they have 0 chance of overshooting, and they make more progress toward 4), and the calculations are born out by the calculation. So if the players are playing correctly then neither player will ever have a score of 1, so none of the states in the second row or second column will be reached. Now, these are reached in the early games, when players are choosing their moves essentially randomly, but once they have switched from "explore" to "exploit" mode, they will only very rarely roll 1 die on their first turn. So the amount of data for these scores forever remains small; so the estimates of probabilities are certainly poor, and even the choice of move can be off.

In fact, the few experiments that are in the sample inputs and outputs for this assignment suggest that this reinforcement learning doesn't necessarily find the right move even in scores that would often be attained if both players played optimally. The reason is that, at an early stage, one move P may be optimal if later in the game the players will play randomly, whereas a different move Q may be optimal if later in the game the players will play optimally. However, because of the way that the shift from explore to exploit is handled, the earlier move may get stuck on P before the later moves decisively make their best choice.

So the most that can be expected is this: If the parameter `M` is chosen to be reasonably large and the number of games is chosen to be very large (the choice of `NGames` depends on the choice of `M`), then generally, with high probability, the program will make the right choices and converge to the correct probabilities. (I'm not sure that this is always the case.) In this particular game, those are the scores $\langle 0, 0\rangle$, $\langle 2, 2\rangle$, $\langle 2, 3\rangle$, $\langle 3, 2\rangle$, and $\langle 3, 3\rangle$. The other positions can't be expected to match. (It is somewhat noteworthy how well they do actually match in this example, except for the

But as the above examples indicate, the convergence of probability estimates can be surprisingly slow, particularly for small scores far from a terminal state. The estimates on the scores $\langle 3, J\rangle$ are mostly within 0.01 of the true value 0.5; but the estimates on the score $\langle 0, 0\rangle$ are 0.5446, 0.4982, and 0.5429, not particularly good estimates of the true value 0.57.

In short, if the results of your experiments don't match mine, or don't match the correct answers, don't worry. That's to be expected. Only part of the answer can be relied on to match at all; and it may not match very closely.

The wrong way to write and test your code

Unless you reliably write bug-free code[1] then the wrong way to proceed with this assignment is to write all the code and then try running it on my examples and see whether your results match mine.

The right way to write and test your code

Rather, you should write a number of modules, and test them, working bottom-up. Personally I wrote six functions:

`chooseFromDist(p)` – Given a probability distribution `p` over the values from 1 to `p.length`, choose a random value. Use the algorithm in the assignment. If you want to test this, pick some test value of `p`; and write a driver that runs it 1000 times, and keep a tally of how many times each value was reached. The frequencies of the different values should more or less correspond to the original distribution `p`.

`rollDice(NDice,NSides)` – Simulate the rolling of `NDice` dice with `NSides` sides.

`chooseDice(Score, LoseCount, WinCount, NDice, M)` – Pick the number of dice to roll, given the parameters, following the formulas in the assignment. You should work out what the numbers f_j and p_j are for some particular value of the above parameters, and then make sure that you are calculating them correctly.

`PlayGame(NDice,NSides,LTarget,UTarget,LoseCount,WinCount,M)` – Play 1 game with the above parameters. The output is new values of `LoseCount` and `WinCount`. You should keep track of the trace of the game as it proceeds (the sequence of score, number of dice rolled, and outcome of the roll), and then make sure that the changes to `LoseCount` and `WinCount` correspond to the trace.

`extractAnswer(WinCount,LoseCount)` – given the final state of the arrays, extract the best move in each state and the probability of winning if you make that move.

`prog3(NDice, NSides, LTarget, UTarget, NGames, M)` Top-level function. Once all the other pieces are working, this is four or five lines of code. Initialize `LoseCount` and `WinCount` to arrays of 0, play the game `NGames` times, and then extract the answer from the final state of `WinCount` and `LoseCount`.

[1] There are such people, but they are rare.