CS 747: Programming Assignment 2

(TA in charge: Akshay Arora)

In this assignment, you will implement algorithms for finding an optimal policy for a given MDP. The first part of the assignment is to apply Linear Programming, based on the formulation presented in class. The second part of the assignment is to implement Howard's Policy Iteration. The third part of the assignment requires you to apply these solvers to a variant of the Gambler's Problem (described in the textbook by Sutton and Barto (2018, see Example 4.3)): in fact your job is essentially to formulate this task as an MDP.

Data

This directory provides a few samples of input and output that you can use to test your code. The directory contains four MDPs encoded as text files, with each file in the following format.

```
Number of states
Number of actions
Reward function
Transition function
Discount factor
Type
```

In these files, and also in the MDPs on which your algorithms will be tested, the number of states S will be an integer greater than 0 and less than 150. Assume that the states are numbered 0, 1, 2, …, (S - 1). Similarly, actions will be numbered 0, 1, 2, …, (A - 1), where A is less than 100. The reward function will be provided over S×A lines, each line containing S entries. Each entry corresponds to R(s, a, s'), wherein state s, action a, and state s' are being iterated in sequence from 0 to (S - 1), 0 to (A - 1), and 0 to (S - 1), respectively. A similar scheme is adopted for the transition function T. Each reward lies between -1 and 1 (both included). The discount factor is a real number in [0, 1]. However, the discount factor will only be set to 1 if the underlying task is episodic. The last field in the file, denoted Type, will either be "continuing" or "episodic". If episodic, it is our convention that the very last state (numbered S - 1) will be a terminal state. The MDP will be such that for every starting state and policy, trajectories will eventually reach the terminal state.

Below is a snippet of python code that is used to generate MDP files.

```
print S
print A

for s in range(0, S):
    for a in range(0, A):
        for sPrime in range(0, S):
            print str(R[s][a][sPrime]) + "\t",

        print "\n",

for s in range(0, S):
    for a in range(0, A):
        for sPrime in range(0, S):
            print str(T[s][a][sPrime]) + "\t",

        print "\n",

print gamma
print type
```

Solution

Given an MDP, your program must compute the optimal value function V* and an optimal policy π* by applying the algorithm that is specified through the command line. Create a shell script called `planner.sh` to invoke your program. The arguments to `planner.sh` will be

- `--mdp` followed by a full path to the input MDP file, and
- `--algorithm` followed by one of `lp` and `hpi`.

Make no assumptions about the location of the

MDP file relative to the current working directory; read it in from the full path that will be provided. The algorithms specified above correspond to Linear Programming and Howard's Policy Iteration, respectively. Here are a few examples of how your planner might be invoked (it will always be invoked from its own directory).

- `./planner.sh --mdp /home/user/temp/data/mdp-7.txt --algorithm lp`
- `./planner.sh --mdp /home/user/mdpfiles/mdp-5.txt --algorithm hpi`

You are free to implement the planner in any programming language of your choice. You are not expected to code up a solver for LP; rather, you can use available solvers as blackboxes (more below). Your effort will be in providing the LP solver the appropriate input based on the MDP, and interpreting its output appropriately. You are expected to write your own code for Howard's Policy Iteration; you may not use any libraries that might be available for the purpose.

Output Format

The output of your planner must be in the following format, and **written to standard output**.

```
V*(0)   π*(0)
V*(1)   π*(1)
.
.
.
V*(S - 1)   π*(S - 1)
```

In the `data` directory provided, you will find four output files corresponding to the MDP files, which have solutions in the format above.

Since your output will be checked automatically, make sure you have nothing printed to stdout other than the S lines as above in sequence. If the testing code is unable to parse your output, you will not receive any marks.

Note:

1. Your output has to be written to the standard output, not to any file.
2. For values, print at least 6 places after the decimal point. Print more if you'd like, but 6 (`xxx.123456`) will suffice.
3. If your code produces output that resembles the solution files: that is, S lines of the form

   ```
   value + "\t" + action + "\n"
   ```

   or even

   ```
   value + " " + action + "\n"
   ```

   you should be okay. Make sure you don't print anything else.

4. If there are multiple optimal policies, feel free to print any one of them.

Gambler's Problem

For a full description of the Gambler's Problem, see Example 4.3 in the textbook by Sutton and Barto (2018). As in the Exercise 4.9, your objective is to prepare plots of the optimal value function for different values of p_h.

Rather than write separate code for the Gambler's Problem, you will merely encode the problem as an MDP, on which your Linear Programming and Policy Iteration algorithms have already been designed to work. You must create a script called `encodeGambler.sh`, which will take p_h as input, and write out an MDP (in the same format as described above) to standard output. We will invoke your program as follows.

```
./encodeGambler.sh ph > mdpFileName
```

Here `ph` will be a number between 0 and 1 (both excluded), and `mdpFileName` will be the full path to a file into which your MDP gets written. Thereafter we will run your policy iteration code on the same MDP, as follows, and manually inspect the values printed out.

```
./planner.sh --algorithm hpi
```

The challenge of this exercise is for you to be able to pose the Gambler's Problem using the MDP format we have adopted. Your formulation must be mathematically correct: that is, optimal values in the MDP must indeed be the Gambler's maximum expected profits from corresponding states. You do have some leeway in terms of deciding what your states, actions, rewards, and transition probabilities must be. If you run into technical difficulties on account of the discount factor or the type of the task, use your understanding of the task to make suitable modeling choices. Make sure you document all these choices in your report.

Apart from fully describing the rationale behind your encoding in your report, include graphs for p_h values of 0.2, 0.4, 0.6, and 0.8. Interpret the results: do the graphs agree with your intuition?

Submission

You will submit three items: (1) working code for your planner, which implements the two different algorithms, (2) working code for encoding the Gambler's Problem as an MDP, and (3) a report describing your MDP encoding, results, and observations on the Gambler's Problem.

Create a directory titled `[rollnumber]`. Place all your source and executable files in this directory. The directory must contain scripts titled `planner.sh` and `encodeGambler.sh`, which must take in the command line arguments specified above, and produce the output also as specified. Also place `report.pdf` in the same directory.

Before you submit, make sure you can successfully run `planner.sh` and `encodeGambler.sh` on the departmental (`sl2`) machines. Provide references to any libraries and code snippets you have utilised (mention them in `report.pdf`). It is okay to use libraries for data structures and for operations such as sorting. You may also use libraries for solving systems of linear equations. However, the logic used for policy improvement, and for translating the given MDP into a linear program, and for encoding the Gambler's Problem, must entirely be code that you have written.

Compress and submit your directory as `[rollnumber].tar.gz`. The directory must contain all the sources, executables, and experimental data, and importantly, your `planner.sh` and `encodeGambler.sh` scripts and `report.pdf` file. Make sure you upload your submission to Moodle by the submission deadline.

Evaluation

Your planner will be tested on a large number of MDPs. Your task is to ensure that it prints out the correct solution (V* and π*) in each case, using each of the algorithms you have been asked to implement. 3 marks each are allotted for the correctness of your Linear Programming and Howard's Policy Iteration algorithms. 2 marks are allotted for the correctness of your encoding of Gambler's Problem, and another 2 marks for your results and observations.

The TAs and instructor may look at your source code to corroborate the results obtained by your program, and may also call you to a face-to-face session to explain your code.

Deadline and Rules

Your submission is due by 11.55 p.m., Sunday, September 9. You are advised to finish working on your submission well in advance, keeping enough time to test it on the `sl2` machines and upload to Moodle. Your submission will not be evaluated (and will be given a score of zero) if it is not received by the deadline.

Before submission, make sure that your code runs for a variety of experimental conditions. Test your code on the `sl2` machines even while you are developing it: do not postpone this step to the last minute. If your code requires any special libraries to run, it is *your* responsibility to get those libraries working on the `sl2` machines (go through the CSE bug tracking system to make a request to the system administrators). Make sure that you upload the intended version of your code to Moodle (after uploading, download your submission and test it on the `sl2` machines to make sure it is the correct version). You will not be allowed to alter your code in any way after the submission deadline. In short: your grade will be completely determined by your submission on Moodle at the time of the deadline. Play safe by having it uploaded and tested at least a few hours in advance.

You must work alone on this assignment. Do not share any code (whether yours or code you have found on the Internet) with your classmates. Do not discuss the design of your solution with anybody else. Do not see anybody else's code or report.

References for Linear Programming

Although you are free to use any library of your choice for LP, we recommend that you use the Python library `PuLP` (https://pythonhosted.org/PuLP/) or the `lp_solve` program (http://lpsolve.sourceforge.net/5.5/). Both of these are already installed on the `sl2` machines.

`PuLP` is convenient to use directly from Python code: here is a short tutorial and here is a reference.

`lp_solve` can be used both through an API and through the command line. Here is a reference and here is an introductory example.