CS 747: Programming Assignment 2

(Prepared by Aaron Jerry Ninan and Adit Akarsh)

In this assignment, you will write code to compute an optimal policy for a given MDP using the algorithms that were discussed in class: Value Iteration, Howard's Policy Iteration, and Linear Programming. The first part of the assignment is to implement these algorithms. Input to these algorithms will be an MDP and the expected output is the optimal value function, along with an optimal policy. You also have to add an optional command line argument for the policy, which evaluates the value function for a given policy instead of finding the optimal policy, and returns the action and value function for each state in the same format.

MDP solvers have a variety of applications. As the second part of this assignment, you will use your solver to find an optimal policy for a batter chasing a target during the last wicket in a game of **cricket**.

This compressed directory contains a `data` directory with sample data for both parts and helper functions to visualize and test your code. Your code will also be evaluated on instances other than the ones provided.

Part 1: MDP Planning

Data

In the `mdp` folder in `data` directory, you are given six MDP instances (3 each for continuing and episodic tasks). A correct solution for each MDP is also given in the same folder, which you can use to test your code.

MDP file format

Each MDP is provided as a text file in the following format.

numStates S
numActions A
end ed1 ed2 ... edn
transition *s1 ac s2 r p*
transition *s1 ac s2 r p*
. . .
. . .
. . .
transition *s1 ac s2 r p*
mdptype *mdptype*
discount *gamma*

The number of states S and the number of actions A will be integers greater than 1. There will be at most 2500 states, and at most 100 actions. Assume that the states are numbered 0, 1, ..., S - 1, and the actions are numbered 0, 1, ..., A - 1. Each line that begins with "transition" gives the reward and transition probability corresponding to a transition, where R(s1, ac, s2) = r and T(s1, ac, s2) = p. Rewards can be positive, negative, or zero. Transitions with zero probabilities are not specified. *mdptype* will be one of `continuing` and `episodic`. The discount factor *gamma* is a real number between 0 (included) and 1 (included). Recall that gamma is a part of the MDP: you must not change it inside your solver! Also recall that it is okay to use gamma = 1 in episodic tasks that guarantee termination; you will find such an example among the ones given.

To get familiar with the MDP file format, you can view and run `generateMDP.py` (provided in the `base` directory), which is a python script used to generate random MDPs. Specify the number of states and actions, the discount factor, type of MDP (episodic or continuing), and the random seed as command-line arguments to this file. Two examples of how this script can be invoked are given below.

- `python generateMDP.py --S 2 --A 2 --gamma 0.90 --mdptype episodic --rseed 0`
- `python generateMDP.py --S 50 --A 20 --gamma 0.20 --mdptype continuing --rseed 0`

Task 1 - MDP Planning Algorithms

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 python file called `planner.py` which accepts the following command-line arguments.

- `--mdp` followed by a path to the input MDP file, and
- `--algorithm` followed by one of `vi`, `hpi`, and `lp`. You must assign a default value out of vi, hpi, and lp to allow the code to run without this argument.
- `--policy` (optional) followed by a policy file, for which the value function V^π is to be evaluated.

The policy file has one line for each state, containing only a single integer giving the action.

Make no assumptions about the location of the MDP file relative to the current working directory; read it in from the path that will be provided. The algorithms specified above correspond to Value Iteration, Howard's Policy Iteration, and Linear Programming, respectively. Here are a few examples of how your planner might be invoked (it will always be invoked from its own directory).

- `python planner.py --mdp /home/user/data/mdp-4.txt --algorithm vi`
- `python planner.py --mdp /home/user/temp/data/mdp-7.txt --algorithm hpi`
- `python planner.py --mdp /home/user/mdpfiles/mdp-5.txt --algorithm lp`
- `python planner.py --mdp /home/user/mdpfiles/mdp-5.txt`
- `python planner.py --mdp /home/user/mdpfiles/mdp-5.txt --policy pol.txt`

Notice that the last two calls do not specify which algorithm to use. For the fourth call, your code must have a default algorithm from `vi`, `hpi`, and `lp` that gets invoked internally. This feature will come handy when your planner is used in Task 2. It gives you the flexibility to use whichever algorithm you prefer. The fifth call requires you to evaluate the policy given as input (rather than compute an optimal policy). You are free to implement his operation in any suitable manner.

You are not expected to code up a solver for LP; rather, you can use available solvers as black-boxes. Your effort will be in providing the LP solver the appropriate input based on the MDP, and interpreting its output appropriately. Use the formulation presented in class. We recommend you to use the Python library `PuLP`. `PuLP` is convenient to use directly from Python code: here is a short tutorial and here is a reference. `PuLP` version 2.4 is installed in the CS 747 docker.

You are expected to write your own code for Value Iteration and Howard's Policy Iteration; you may not use any custom-built libraries that might be available for the purpose. You can use libraries for solving linear equations in the policy evaluation step but must write your own code for policy improvement. Recall that Howard's Policy Iteration switches **all** improvable states to some improving action; if there are two or more improving actions at a state, you are free to pick anyone.

It is certain that you will face some choices while implementing your algorithms, such as in tie-breaking, handling terminal states, and so on. You are free to resolve in any reasonable way; just make sure to note your approach in your report.

Output Format

The output of your planner must be in the following format and **written to standard output, for both modes of operation.** Note that, for policy evaluation, V* will be replaced by V^π.

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

In the `data/mdp` directory provided, you will find 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.

You are given a python script to verify the correctness of your submission format and solution: `autograder.py`. The following are a few examples that can help you understand how to invoke this script.

- `python autograder.py --task 1` --> Tests the default algorithm set in `planner.py` on the all the MDP instances given to you in the `data/mdp` directory.
- `python autograder.py --task 1 --algorithm all` --> Tests all three algorithms + default algorithm on the all the MDP instances give to you in the `data/mdp` directory.

- `python autograder.py --task 1 --algorithm vi` --> Tests only value iteration algorithm on the all the MDP instances given to you in the `data/mdp` directory.

The script assumes the location of the `data` directory to be in the same directory. Run the script to check the correctness of your submission format. Your code should pass all the checks written in the script. You will be penalised if your code does not pass all the checks.

Your code for any of the algorithms should not take more than one minute to run on any test instance.

Part 2: Cricket: The last wicket

The Game

The following problem is based on the game of Cricket. In Cricket, there are two teams, each with 11 players- the batting and the bowling team, which play on a pitch (between 2 wickets) located at the centre of the ground. The batting side scores runs by striking the ball bowled at the wicket with the bat and then running between the wickets, while the bowling and fielding side tries to prevent this (by preventing the ball from leaving the field, and getting the ball to either wicket) and dismiss each batter (so they are "out"). At any instance there are 2 players (striker and non-striker) from the batting team, and 11 players from the bowling team (1 bowler, 10 fielders) on the field. After every 6 balls (also called *over*) the stiker and non-striker swap their positions: that is, the non-striker becomes a striker and vice-versa. A striking battter can either get dismissed (or *out*) or can score one of 0, 1, 2, 3, 4, 6 runs. To score 0, 1, 2, or 3 runs the batters need to swap their positions respective amount of times in the same ball. A striking batter can hit a 4 or 6 if the ball crosses the boundary (6 incase the ball crosses aerially).

Consider two batters A and B. In this task we aim to find the optimal policy for batter A, assuming we have no control over the actions of batter B: that is, batter A is the agent and batter B along with rest of game dynamics is part of the environment. A is a middle-order batter at the last wicket, along with a tail-ender 'B'. Additionally, consider that B can only either get out or score 0/1 runs. As part of the environment, we can fix a parameter "q" indicating the *degree of weakness* of B (a detailed description is given in the next section) The batting team still has to get T runs (T <= 30) in O balls (O <= 15). We can formulate this problem as an MDP. The set of states is encoded in the form bbrr, where bb and rr are 2-digit numbers representing the number of balls left, and the number of runs to score to reach the target. Single digit numbers have a 0 attached to the left to reach 2 digits (eg - 0701 for 1 run to score in 7 balls).

State File Format

You are provided with a file *cricket_states.py* for generating the ordered list of states that the batter can face in the game, given the initial runs to score and the balls remaining.

Actions and outcomes

The set of actions available to the batter are {0, 1, 2, 4, 6}.
0- Defend
1- Attempt a single run
2- Attempt 2 runs
4- Attempt a boundary (4 runs)
6- Attempt a six

The set of outcomes at each ball are {-1, 0, 1, 2, 3, 4, 6}.
-1- Out; the game ends
0- 0 runs
1- 1 run
2- 2 runs
3- 3 runs
4- 4 runs
6- 6 runs

At the beginning of every over, the strike rotates. And for 1 and 3 runs scored, the strike changes. If both the over changes and 1 or 3 runs are scored, the strike remains with the same batter. The over can be assumed to change at every ball divisible by 6 (that is, bb = 12, 06).

Player parameters

Player A has parameters given in the form of text files-
action runs_probability (-1, 0, 1, 2, 3, 4, 6)
0 0.01 0.69 0.3 0 0 0 0
1 0.02 0.5 0.48 0 0 0 0
2 0 0.2 0.3 0.5 0 0 0
4 0.2 0.1 0.1 0 0 0.55 0.05
6 0.4 0.1 0 0 0 0.2 0.3

Player B has only one parameter- q, a real number in [0,1]. It gives the probabilities of B hitting a [-1, 0, 1] as [q, (1-q)/2, (1 - q)/2] respectively.

Policy File Format

A policy specifies the complete behaviour of any one of the players. Policy files will have the state having the information of balls remaining and runs to score *bbrr*, and the optimal action to be taken by 'A', and the optimal value function (V*) for the state (again, print at least 6 places after the decimal point), separted by a space. Example-

```
1506 4 0.453213
1505 4 0.501322
1504 2 0.575312
1503 2 0.775312
```

Task 2 - MDP for Cricket Game

In this task, your objective is to find an optimal policy for batter 'A' with for the above game of cricket at the last wicket. Assume that a win earns a score of 1, every other outcome (draw, loss) earns 0. Thus, in the optimal policy, the batter will choose actions so as to maximise the probability of winning.

Rather than write a solver from scratch, your job is to translate the above task into an MDP, and use the solver you have already created in Task 1.

Your first step is to write an encoder script which takes as input the player parameters and the set of states and encodes the game into an MDP (use the same format as described above).

Your next step is to take the output of the encoder script and use it as input to `planner.py`. As usual, `planner.py` will provide as output an optimal policy and value function.

Finally, create a python file called `decoder.py` that will take the output from `planner.py` and convert it into a policy file in the format given above for the cricket game, having the same format as stated above. Thus, the sequence of encoding, planning, and decoding results in the computation of an optimal policy for a player given player parameters and the number of balls and runs.

Here is the sequence of instructions that we will execute to obtain your optimal counter-policy.

```
python encoder.py --states statefilepath --parameters p1_parameters --q q > mdpfile
python planner.py --mdp mdpfile > value_and_policy_file
python decoder.py --value-policy value_and_policy_file --states statesfilepath > policyfile
```

As stated earlier, note that the planner must use its default algorithm. You can use `autograder.py` to verify that your encoding and decoding scripts produce output in the correct format, and correctly for the two provided sample player 1 parameters and q = 0.25. Here is how you would use it.

```
python autograder.py --task 2
```

The eventual output `policyfile` will be evaluated by us (using our own code) to check if it achieves optimal score from every valid state.

Analysis

To check the effects of the environment parameters, you have to plot 3 graphs. Each graph should have two lines- one for the optimal policy you obtain from the above code, and one for the arbitrary policy we have provided in `rand_pol.txt`. This policy was generated at random, with no particular logic in mind.

1. Fix state (15 balls, 30 runs) and plot win probability (playing as per the policy) vs. B's "strength" (q, varied from 0 to 1). Note that the win probability from any state is given by the value of the value function at that state for the policy. For the policy we have provided, you can use the policy evaluation mode from task 1.
2. Fix the number of balls as 10 balls, q as 0.25 and vary runs to score as 20 runs, 19 runs, 18 runs,...,1 run. Plot win probability (playing as per policy) vs. runs. Again, plot for both policies.
3. Fix the number of runs as 10 runs to score, q as 0.25 and vary balls remaining as 15 balls, 14 balls, ..., 1 ball. Plot win probability (playing as per policy) vs. balls. Again, plot for both policies.

Attach these three graphs in the report along with observations. Do the trends match with your intuition? Is your optimal policy able to produce better results than the baseline wehave provided?

Submission

Prepare a short `report.pdf` file, in which you put your design decisions, assumptions, and observations about the algorithms (if any) for Task 1. Also describe how you formulated the MDP for the Cricket problem: that is, for Task 2. Finally, include the 3 graphs for Task 2 with your observations.

Place all the files in which you have written code in a directory named `submission`. Tar and Gzip the directory to produce a single compressed file (submission.tar.gz). It must contain the following files.

1. `planner.py`
2. `encoder.py`
3. `decoder.py`
4. `report.pdf`
5. `references.txt`
6. Any other files required to run your source code

Submit this compressed file on Moodle, under Programming Assignment 2.