CS 747: Programming Assignment 2

(Prepared by Santhosh)

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.

MDP solvers have a variety of applications. As the second part of this assignment, you will use your solver to find the shortest path between a given start state and an end state in a maze.

Data

The `data` directory in this compressed directory contains 2 folders: `mdp` and `maze`. In the `mdp` folder, 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. In the `maze` folder, you are given 10 maze instances that you can use to test your code for the maze problem. Your code will also be evaluated on instances other than the ones provided.

MDP file format

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

numStates S
numActions A
start st
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, and at most 100. 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.

st is the start state, which you might need for Task 2 (ignore for Task 1). ed1, ed2,..., edn are the end states (terminal states). For continuing tasks with no terminal states, this list is replaced by -1.

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.

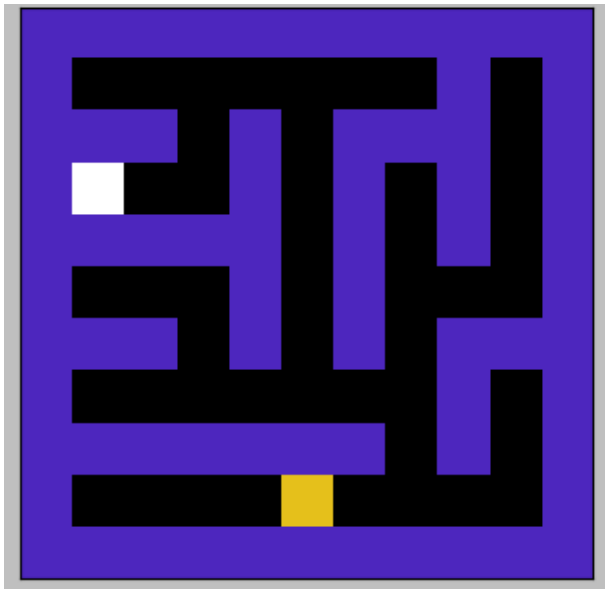
- `python generateMDP.py --S 2 --A 2 --gamma 0.90 --mdptype episodic --rseed 0`
- `python generateMDP.py --S 50 --A 20 --gamma 0.20 --mdptype continuing --rseed 0`

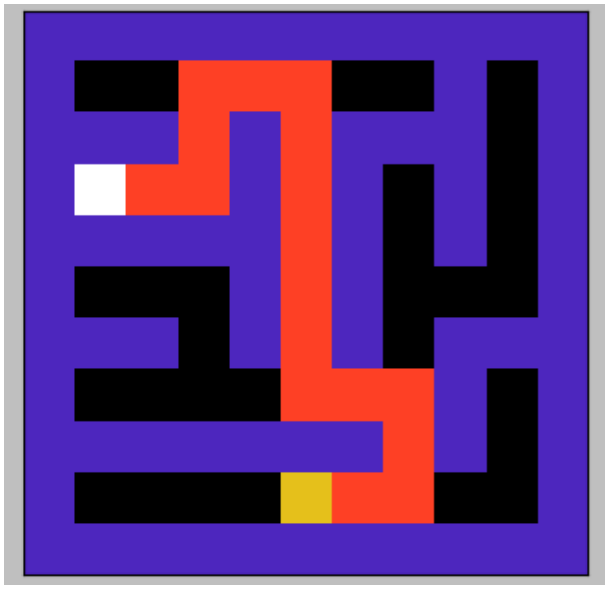
Maze file format

Each maze is provided in a text file as a rectangular grid of 0's, 1's, 2, and 3's. An example is given here along with the visualisation.

1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 1 0 1
1 1 1 0 1 0 1 1 1 0 1
1 3 0 0 1 0 1 0 1 0 1
1 1 1 1 1 0 1 0 1 0 1
1 0 0 0 1 0 1 0 0 0 1
1 1 1 0 1 0 1 0 1 1 1
1 0 0 0 0 0 0 0 1 0 1
1 1 1 1 1 1 1 0 1 0 1
1 0 0 0 0 2 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1

Here 0 denotes an empty tile, 1 denotes an obstruction/wall, 2 denotes the start state and 3 denotes an end state. In the visualisation below, the white square is the end position and the yellow one is the start position.





The figure on the right shows the shortest path.

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`.

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`

You are not expected to code up a solver for LP; rather, you can use available solvers as black-boxes (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 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.

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/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.

Task 2 - Solving a maze using MDPs

In this task, your objective is to find the shortest path from start to end in a specified maze. The idea is to piggyback on the planner code you have already written in Task 1.

Note: In this task, assume that any **invalid move doesn't change the state** e.g., a right move doesn't change the state if there's a wall on the immediate right of the current cell.

Your first step is to encode the maze as an MDP (use the same format as described above). Then you will use `planner.py` to find an optimal policy. Finally, you will simulate the optimal policy on the maze in a deterministic setting to extract a path from start to the end. Note that this path also corresponds to the shortest possible path from start to end.
Output the path as: A0 A1 A2 . . . .
Here "A0 A1 A2 . . ." is the sequence of moves taken from the start state to reach the end state along the simulated path. Each move must be one of N (north), S (south), E (east), and W (west). See, for example, `solution10.txt` in the `maze` directory, for an illustrative solution.

To visualise the maze, use this command. `python visualize.py gridfile`

To visualise your solution, use this command. `python visualize.py gridfile pathfile`

Create a python file called `encoder.py` that will encode the maze as an MDP and output the MDP. The code should run as: `python encoder.py --grid gridfile > mdpfile`

We will then run `python planner.py --mdp mdpfile --algorithm vi > value_and_policy_file`.

Also, create a python source file called `decoder.py` that will simulate the optimal policy and output the path taken between the start and end state given the file `value_and_policy_file` and the `gridfile`. The output format should be as specified above. The script should run as follows.

```
python decoder.py --grid gridfile --value_policy value_and_policy_file
```

Submission

There are two python scripts given to verify the correctness of your submission format and solution: `PlannerVerifyOutput.py` and `MazeVerifyOutput.py`. Both of these scripts accept `--algorithm` as a command-line argument. The following are a few examples that can help you understand how to invoke these scripts.

- `python PlannerVerifyOutput.py` --> Tests all three algorithms on the all the MDP instances given to you in the `data/mdp` directory.
- `python PlannerVerifyOutput.py --mdp --algorithm vi` --> Tests only value iteration algorithm on the all the MDP instances given to you in the `data/mdp` directory.
- `python MazeVerifyOutput.py --mdp --algorithm vi` --> Tests all the maze instances that are given in `data/maze` directory. Here, `--algorithm` specifies the algorithm that you would like to run on the MDP generated by `encoder.py` (we will check using value iteration).

These scripts assume the location of the `data` directory to be in the same directory. Run these scripts 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.

Prepare a short `report.pdf` in which you put your design decisions, assumptions, and observations about the algorithms (if any). Also mention how you formulated the MDP for the maze problem.

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.

Evaluation

7 marks are reserved for Task 1 (2 marks each for the correctness of your Value Iteration and Howard's Policy Iteration algorithms, and 3 marks for Linear Programming).

3 marks are allotted for the correctness of Task 2.

We shall verify the correctness by computing and comparing optimal policies for a large number of unseen MDPs. If your code fails any test instances, you will be penalised based on the nature of the mistake made.

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., Friday, October 23. Finish working on your submission well in advance, keeping enough time to generate your data, compile the results, and upload to Moodle.

Your submission will not be evaluated (and will be given a score of zero) if it is not uploaded to Moodle by the deadline. Do not send your code to the instructor or TAs through any other channel. Requests to evaluate late submissions will not be entertained.

Your submission will receive a score of zero if your code does not execute on cs747 docker container. To make sure you have uploaded the right version, download it and check after submitting (but before the deadline, so you can handle any contingencies before the deadline lapses).

You are expected to comply with the rules laid out in the "Academic Honesty" section on the course web page, failing which you are liable to be reported for academic malpractice.

References for Linear Programming

For the Linear Programming part of the assignment, 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.