CS747 - Programming Assignment 2

Total Marks: 12

(Prepared by Anvay Shah)

In this assignment you will implement Howard's Policy Iteration and Linear Programming to solve for the optimal policy and value function in a Markov Decision Problem (MDP). The first part of the assignment requires you to implement the algorithms and evaluate them on a set of test MDPs. In the second part, you will use the algorithms you implemented to solve a given MDP modelling problem. Lastly, you will be required to write a report detailing your approach and results.

This compressed directory contains a data directory with sample data for both parts and helper functions to visualise and test your code. Your code will also be evaluated on instances other than the ones provided. All the code you write for this assignment must be in Python 3.9.6 (same version used in Programming Assignment 1). If necessary, you can install Python 3.9.6 for your system from here. Your code will be tested using a python virtual environment. To set up the virtual environment, follow the instructions provided in virtual-env.txt which is included in the compressed directory linked above.

Task 1: MDP Planning (5 Marks)

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.
- `--algorithm` followed by one of hpi and lp. You must assign a default value out of 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 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/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 third call, your code must have a default algorithm from 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 fourth call requires you to evaluate the policy given as input (rather than compute an optimal policy). You are free to implement this operation in any suitable manner.

You are not expected to code up a solver for LP; rather, you can use available solvers as blackboxes. 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 require 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 included in the requirements.txt file for the environment.

You are expected to write your own code for 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 any one.

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 them in any reasonable way; just make sure to note down your approach in your report.

Data

In the `mdp` folder in `data` directory, you are given eight MDP instances (4 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 10,000 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. You can assume that there will be at most 350,000 transitions with non-zero probabilities in the input MDP. 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`

Output File 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)
```

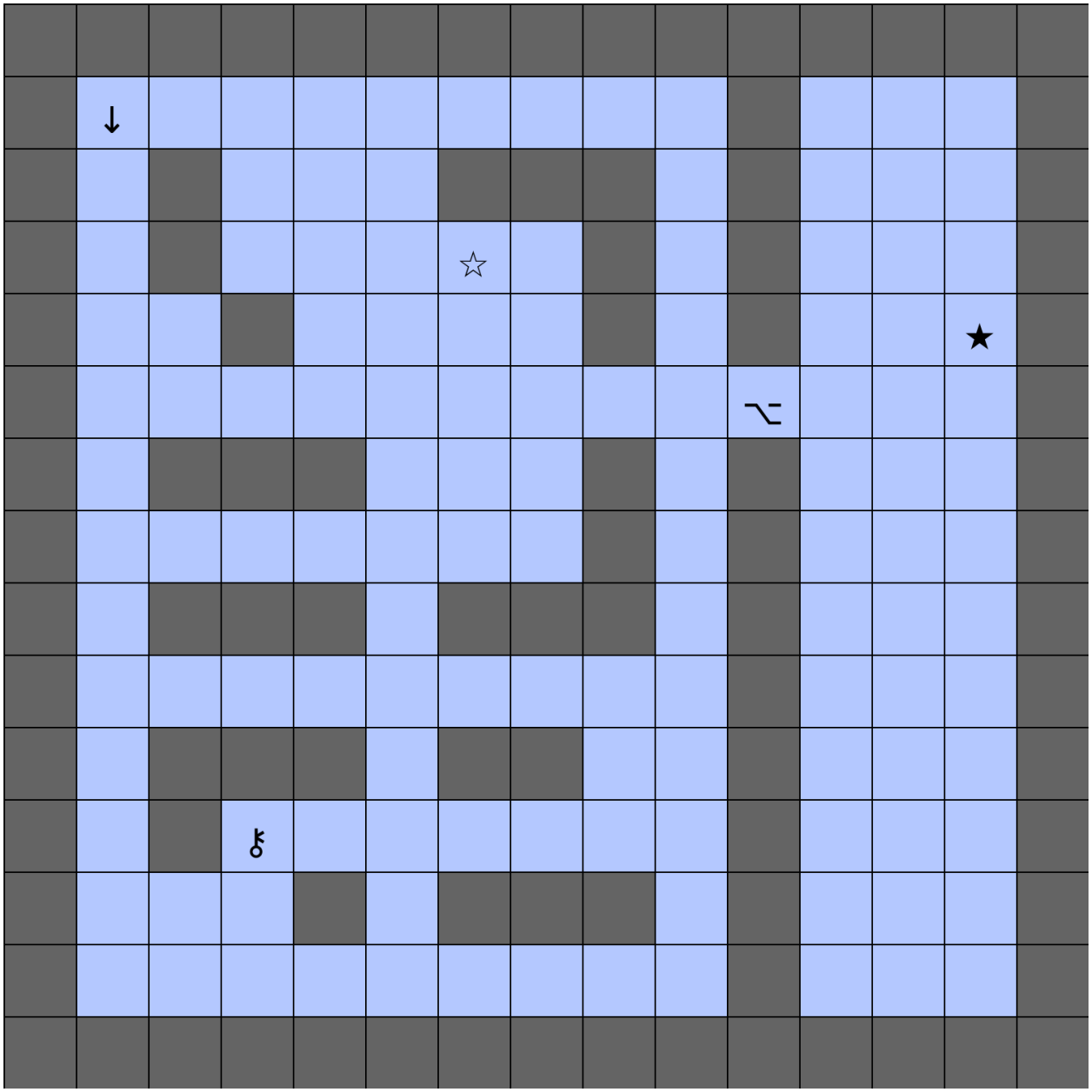
In the data/mdp directory provided, you will find output files corresponding to the MDP files, which have solutions in the format above. Since your output will be checked automatically, make sure you have nothing printed to stdout other than the S lines as above in sequence. If the testing code is unable to parse your output, you will not receive any marks.

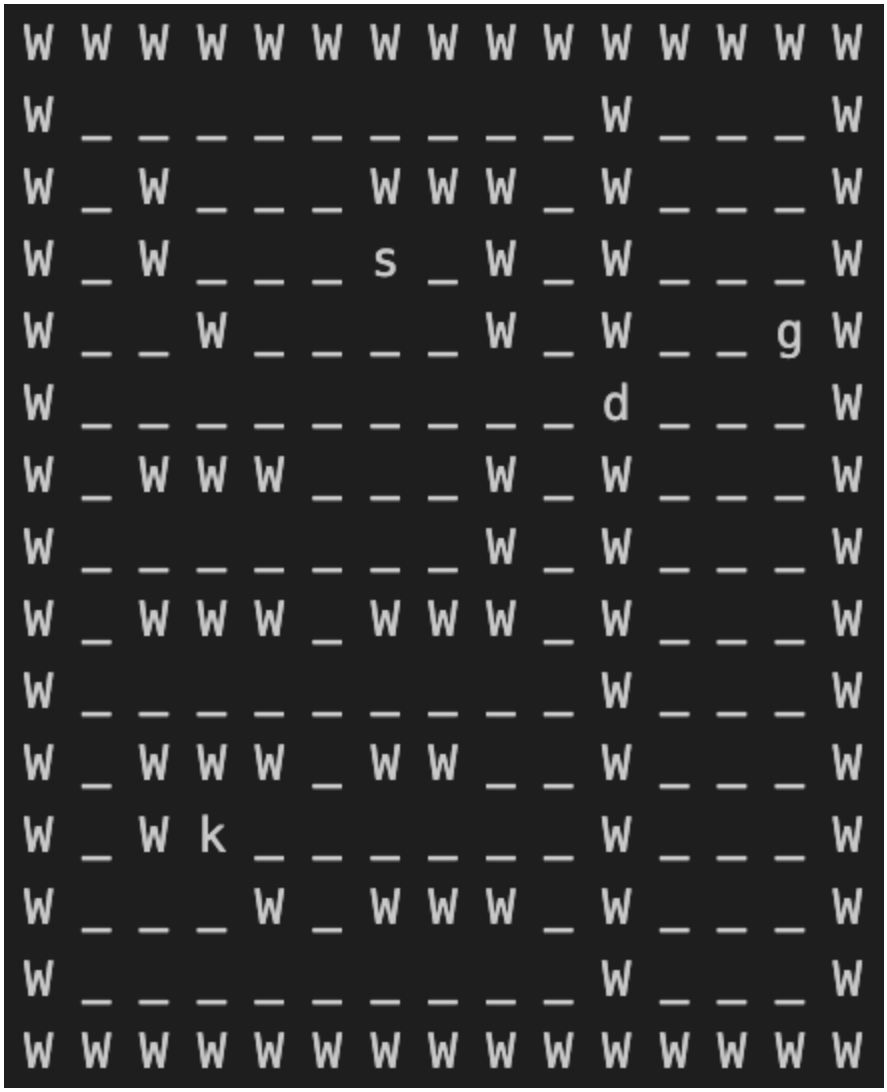
Note: Your output has to be written to the standard output, not to any file. For values, print at least 6 places after the decimal point. Print more if you'd like, but 6 (xxx.123456) will suffice. If your code produces output that resembles the solution files: that is, S lines of the form value + "\t" + action + "\n" or even value + " " + action + "\n" you should be okay. Make sure you don't print anything else. 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.

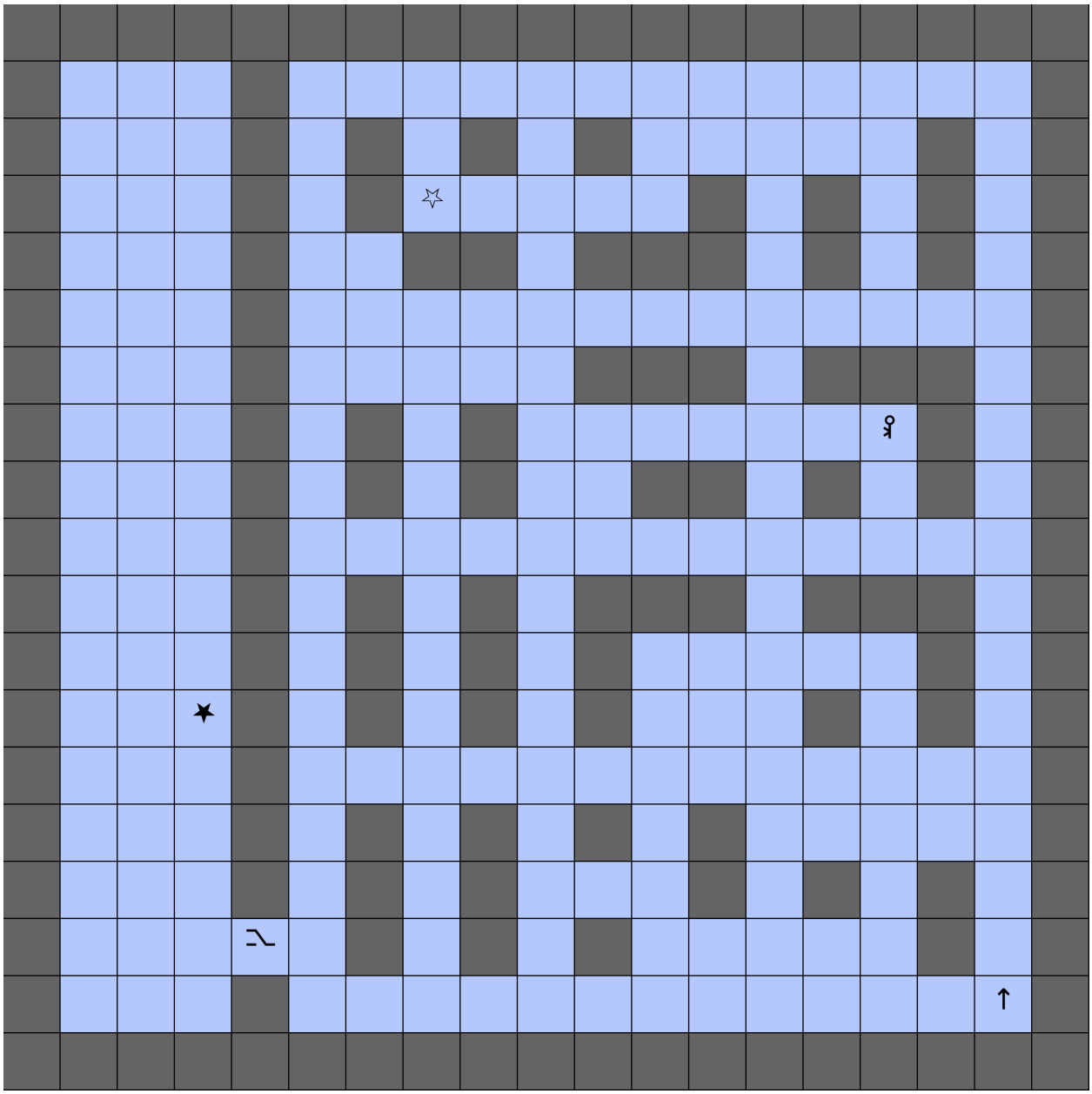
`python3 autograder.py --task 1` --> Tests the default algorithm set in planner.py on the all the MDP instances given to you in the data/mdp directory.
`python3 autograder.py --task 1 --algorithm all` --> Tests all three algorithms + default algorithm on the all the MDP instances give to you in the data/mdp directory.
`python3 autograder.py --task 1 --algorithm pi` --> Tests only policy iteration algorithm on the all the MDP instances given to you in the data/mdp directory.

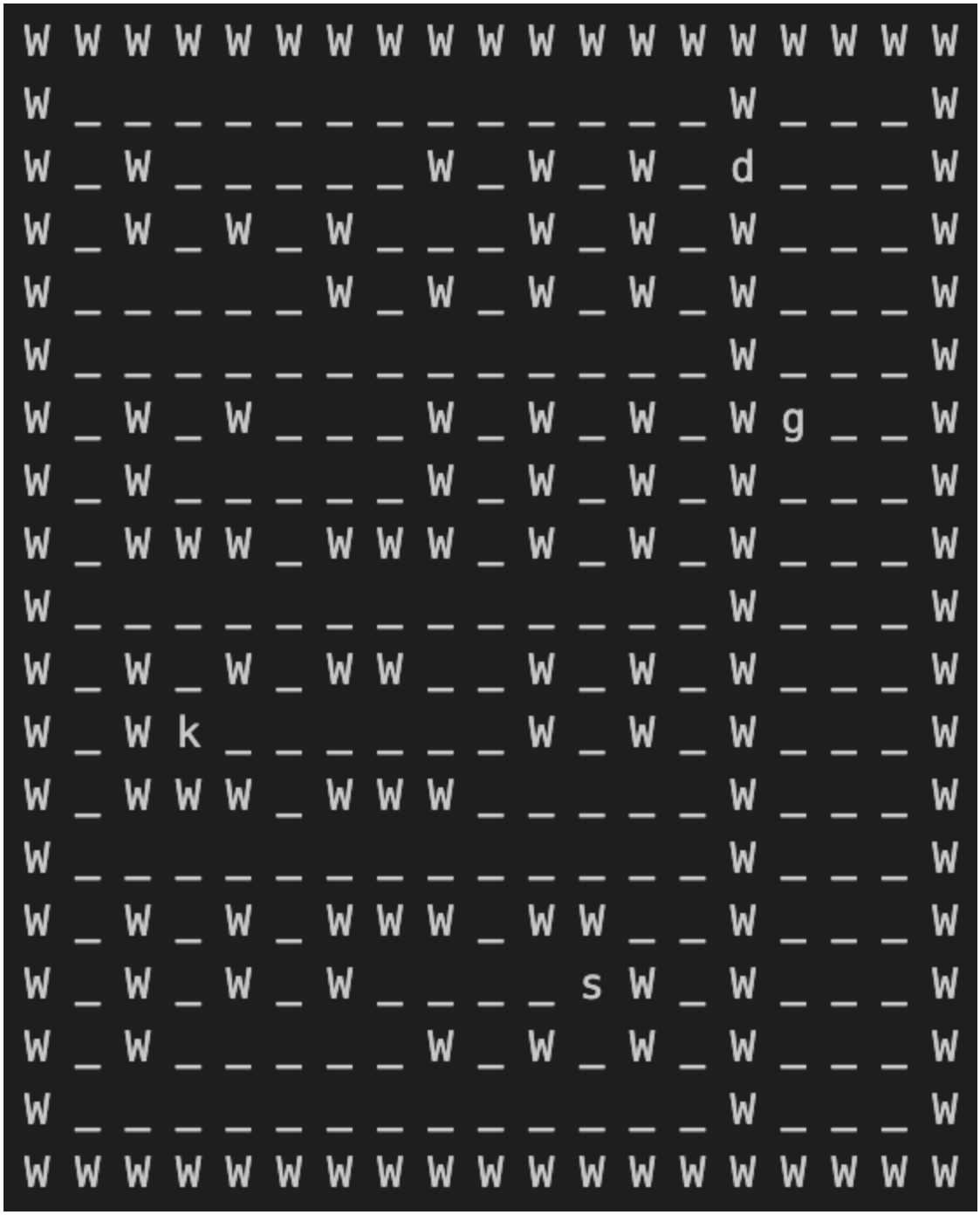
The script assumes the location of the data directory to be in the same directory. Run the script to check the correctness of your submission format. Your code should pass all the checks written in the script. You will be penalised if your code does not pass all the checks.

Task 2: MDP Modeling: Navigating an Icy Gridworld (5 Marks)




W W W W W W W W W W W W W W W
W _ _ _ _ _ _ _ _ _ W _ _ _ W
W _ W _ _ _ W W W _ W _ _ _ W
W _ W _ _ _ s _ W _ W _ _ _ W
W _ _ W _ _ _ _ W _ W _ _ g W
W _ _ _ _ _ _ _ _ _ d _ _ _ W
W _ W W W _ _ _ W _ W _ _ _ W
W _ _ _ _ _ _ _ W _ W _ _ _ W
W _ W W W _ W W W _ W _ _ _ W
W _ _ _ _ _ _ _ _ _ W _ _ _ W
W _ W W W _ W W _ _ W _ _ _ W
W _ W k _ _ _ _ _ _ W _ _ _ W
W _ _ _ W _ W W W _ W _ _ _ W
W _ _ _ _ _ _ _ _ _ W _ _ _ W
W W W W W W W W W W W W W W W





Goal

MDPs are general enough to model various sequential decision-making tasks. If we can solve MDPs, we can thereby find optimal solutions to other tasks. Here is one example - an agent is looking to navigate in a gridworld. The agent's task is decribed for you. Your aim is to first translate the agent's grid into an MDP. You will then use your solver from task 1 to find an optimal policy for this MDP. The optimal policy is then again to be integrated into the gridworld

environment to guide the agent to act optimally inside it.

An agent must navigate an icy gridworld (randomly generated, examples shown above) and reach the goal square (★) in the lowest expected number of time steps. The agent is born in a start square (☆), pointing in one of four directions (→, ←, ↑, ↓). The grey squares are walls and are impervious to the agent. The light blue squares are the only squares the agent is permitted to navigate through. These squares represent the icy floor, which cause the agent to stochastically slide a variable number of steps each time it tries to move forward (rules specified below). Before reaching the goal, the agent will encounter a locked door (⌥). The gridworld is generated such that the wall containing the door seperates the start and goal squares. In order to open this door the agent must locate a key (⚷). The key is "located" simply by reaching the cell with the key; once that is done, the agent has the key in its posession for all time steps afterwards in the episode. Once equipped with the key, the agent will be able to pass through the door. Your task is to model this gridworld as an MDP and solve it using the planner you implemented in Task 1.

You have to implement an encoder and decoder for this gridworld.

The file `encoder.py` must take the gridworld file as a command line argument `--gridworld` and print to standard output an MDP in a format that can be inputted to `planner.py` from task 1. The gridworld file will resemble the files present in `data/gridworld/`. However, the gridworld file will **not** have the agent present within it, whereas test cases (described below) will have the agent initialised in some cell within the grid world and facing some direction.

The file `decoder.py` must take three command line arguments, `--mdp`, `--value-policy`, and `--gridworld`. The value-policy file is the output of the planner, and the MDP file is the output of the encoder. The gridworld file is a test case file, such as the files present in `data/test/`. Each test case contains 5 snapshots of a gridworld with the agent in a certain square (see examples in compressed directory). If the key has already been picked up by the agent, the key will not be present in the gridworld. The decoder must print to standard output space-separated integers, where each integer represents the optimal action the agent must take in the corresponding snapshot. If there are multiple optimal actions, print any one. The actions are numbered 0, 1, 2, and 3.

Your encoder and decoder must work for all randomly generated gridworld instances. You are given 50 sample test cases that you can test your code on using the command `python3 autograder.py --task 2`.

The input gridworld is a text file with a 2D grid of characters (shown in image above). Three example files are given in the `data/gridworld/` directory. The grid will have the following characters:

- 's': Start square

- 'g': Goal square
- 'd': Locked door
- 'k': Key
- '>': Agent facing right
- '<': Agent facing left
- '^': Agent facing up
- 'v': Agent facing down
- 'W': Wall
- '_': Icy floor

You can assume that the input grid world will not have more than 25 cells on each side. The compressed directory contains a file `gridword.py` which contains a Gridworld class with functions for generating a random gridworld, for loading and saving a gridworld, and for visualising an agent's position in the gridworld. You may use this class in your code. You may also use the file `image_gen.py` to generate png images of the gridworlds, as used in this document.

Given below are the rules of the gridworld.

Actions

The agent must select one of four actions at each time step.

1. Move forward.
2. Turn left.
3. Turn right.
4. Turn around (180 degrees).

Movement

Due to the icy floor of the gridworld, the agent's movements are stochastic.

- When the agent attempts to move forward, the agent will slide forward by 1, 2 or 3 steps, with the probabilities p(1) = 0.5, p(2) = 0.3, p(3) = 0.2. If the agent is constrained by a wall n steps away in the forward direction ($n \leq 2$), all probabilities of reaching squares more than n steps away are added to the probability of reaching the nth square. For example, if it is not possible to move 3 steps forward, p(1) = 0.5, p(2) = 0.5. If it is not possible to move even 1 step forward, the agent remains in the same state.
- When the agent attempts to turn left, the agent turns left with probability 0.9, and turns around with probability 0.1.
- When the agent attempts to turn right, the agent turns right with probability 0.9, and turns around with probability 0.1.

- When the agent attempts to turn around, the agent turns around with probability 0.8 and turns left or right with probability 0.1 each.

Note that each grid world task is encoded into its own MDP; that is, you do not have to create a single MDP that works for all grid world tasks. Think carefully about how to define states, actions, transitions, rewards, and values so that an optimal policy for the MDP will indeed result in the agent minimising the expected number of steps to reach the goal.

Report (2 marks)

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 Task 2. Also note down any other interesting aspects that you encountered while working on the assignment.

Submission

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

Note that you must also include a references.txt file if you have referred to any resources while working on this assignment (see the section on Academic Honesty on the course web page). Submit this compressed file on Moodle, under Programming Assignment 2.

Evaluation

5 marks are reserved for Task 1 (2 marks for Howard's Policy Iteration, Linear Programming each, and 1 mark for the policy evaluation). 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.

5 marks are allotted for the correctness of Task 2. We will run the encoder-planner-decoder sequence on randomly generated gridworlds. We will check if the agent's actions are optimal in the gridworlds. If your code fails any test instances, you will be penalised based on the nature of the mistake made.

2 marks are reserved for your report.

Deadline and Rules

Your submission is due by 11.59 p.m., Tuesday, March 18, 2025. Finish working on your submission well in advance, keeping enough time to test your code, 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 using the given python virtual environment. To make sure you have uploaded the right version, download it and check after submitting (but well 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.
