Assignment 6: Markov Decision Processes and Computing Utility Values

CSE 415: Introduction to Artificial Intelligence
The University of Washington, Seattle, Autumn 2019

The search theme continues here, except that now our agents operate in a world in which actions may have uncertain outcomes. The interactions are modeled probabilistically using the technique of Markov Decision Processes. Within this framework, the assignment focuses on two approaches to having the agent maximize its expected utility: (A) by a form of planning, in which we assume that the parameters of the MDP (especially the transition model T and the reward function R) are known, which takes the form of *Value Iteration*, and (B) by an important form of reinforcement learning, called *Q-learning*, in which we assume that the agent does not know the MDP parameters. Part A is required and worth 100 points, while Part B is for 10 points of extra credit.

NOTE: This assignment requires the use of the IDLE integrated development environment, unless you wish to spend a substantial amount of extra time, on your own, configuring your PyCharm or other IDE to use Tkinter. Reports of people who have tried to use Tkinter with other IDEs suggest that a lot of time can be wasted figuring out workarounds for different operating systems, Python versions, and IDE versions. Our advice is, if you already use IDLE, stick with it for this assignment, and if you don't already use IDLE, try it out for this assignment. It's easier to learn than almost any other other Python IDE. IDLE ships with the standard Python distribution from Python.Org, so it is probably already installed on your system. If you don't like IDLE for editing, you could possibly edit the code in another tool, and then reload it into IDLE to run it, but that would feel inconvenient for most of us.

The particular MDPs we are working with in this assignment are variations of a "TOH World", meaning Towers-of-Hanoi World. We can think of an agent as trying to solve a TOH puzzle instance by wandering around in the state space that corresponds to that puzzle instance. If it solves the puzzle, it will get a big reward. Otherwise, it might get nothing, or perhaps a negative reward for wasting time. In Part A, the agent is allowed to know the transition model and reward function. In Part B (optional), the agent is not given that information, but has to learn it by exploring and seeing what states it can get to and how good they seem to be based on what rewards it can get and where states seem to lead to.

Due Friday, November 15 via Canvas at 11:59 PM.

What to Do. Download the starter code. Start up the IDLE integrated development environment that ships with the standard distribution of Python from Python.org. In IDLE,

open the file TOH_MDP.py. Then, using the Run menu on this file's window, try running the starter code by selecting "Run module F5". This will start up a graphical user interface, assuming that you are running the standard distribution of Python 3.6 or Python 3.7, which includes the Tkinter user-interface technology. You can see that some menus are provided. Some of the menu choices are handled by the starter code, such as setting some of the parameters of the MDP. However, you will need to implement the code to actually run the Value Iteration and, if you choose to do the optional Part B, the Q-Learning.

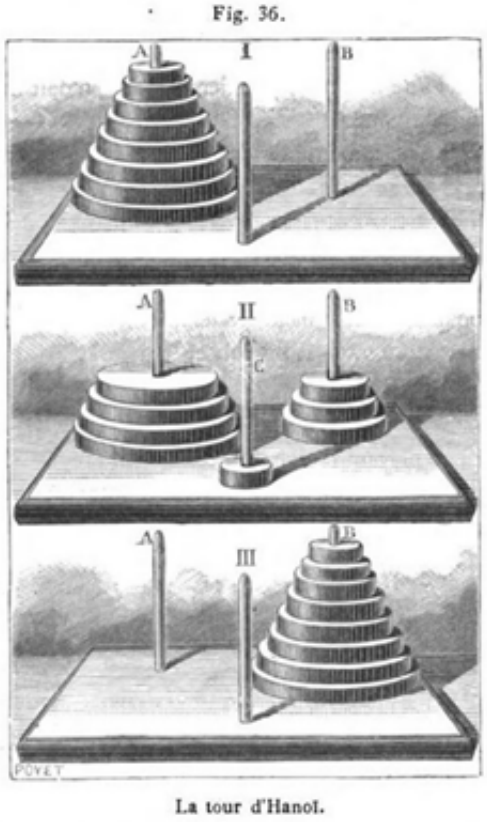

Figure 1. An early description of the problem "La Tour d'Hanoi" from the 1895 book by Edouard Lucas. Your program will determine an optimal policy for solving this puzzle, as well as compute utility values for each state of the puzzle.

PART A: Implement Value Iteration (100 points).

There is a starter file called YourUWNetID_VI.py. Rename the file according to this format and your own UWNetID. Also, change the name within TOH_MDP.py where this file is imported.

Complete the implementation of the functions there:

1. `one_step_of_VI(S, A, T, R, gamma, Vk)`

which can compute 1 iteration of Value Iteration from the given MDP information plus the current state values, which are provided in a dictionary Vk whose keys are states and whose values are floats.

2. `return_Q_values(S, A)`
   which should return the Q-state values that were computed during the most recent call to one_step_of_VI. See the starter code for more details.

3. `extract_policy(S, A)`
   Using the most recently computed Q-state values, determine the implied policy to maximize expected utility. See the starter code for more details.

4. `apply_policy(s)`
   Return the action that your current best policy implies for state s.

A Sample Test Case. Once your implementation is complete, you should be able to duplicate the following example display by setting the following parameters for your MDP and VI computation: from the File menu: "Restart with 2 disks"; from the MDP Noise menu: "20%"; from the MDP Rewards menu: "One goal, R=100"; again from the MDP Rewards menu: Living R= +0.1; from the Discount menu: "γ = 0.9"; from the Value Iteration menu: "Show state values (V) from VI", and "100 Steps of VI" and finally "Show Policy from VI."

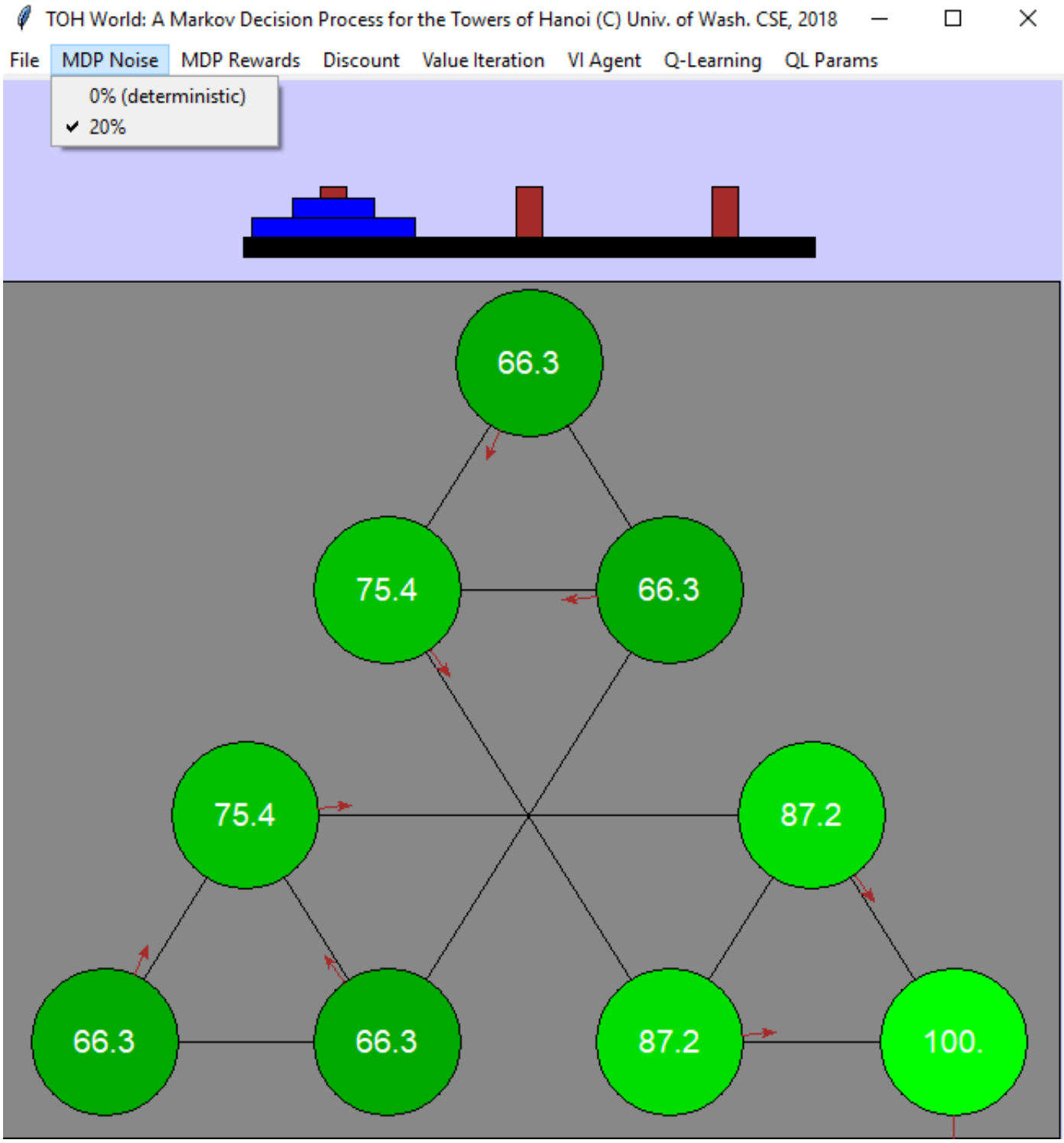

Figure 2. A sample of computed state values. The policy computed from the associated Q values (not shown here) is displayed using red arrows.

Report for Part A.

Create a PDF document that includes the following questions (1a, 1b, etc.) but also their answers. The document should start with this:

```
Assignment 6 Report for Part A
(your name)
```

Name the file using this naming pattern: YourUWNedID_A6_Report.pdf. The following tells you what to do and what questions to answer

1. Using the menu commands, set up the MDP for TOH with 3 disks, no noise, one goal, and living reward=0. The agent will use discount factor 1. From the Value Iteration menu select "Show state values (V) from VI", and then select "Reset state values (V) and Q values for VI to 0".

   Use the menu command "1 step of VI" as many times as needed to answer these questions:

   1a. How many iterations of VI are required to turn 1/3 of the states green? (i.e., get their expected utility values to 100).

   1b. How many iterations of VI are required to get all the states, including the start state, to 100?

   1c. From the Value Iteration menu, select "Show Policy from VI". (The policy at each state is indicated by the outgoing red arrowhead. If the suggested action is illegal, there could still be a legal state transition due to noise, but the action could also result in no change of state.) Describe this policy. Is it a good policy? Explain.

2. Repeat the above setup except for 20% noise.

   2a. How many iterations are required for the start state to receive a nonzero value.

   2c. At this point, view the policy from VI as before. Is it a good policy? Explain.

   2d. Run additional VI steps to find out how many iterations are required for VI to converge. How many is it?

   2e. After convergence, examine the computed best policy once again. Has it changed? If so, how? If not, why not? Explain.

3. Repeat the above setup, including 20% noise but with 2 goals and discount = 0.5.

   3a. Run Value Iteration until convergence. What does the policy indicate? What value does the start state have? (start state value should be 0.82)

   3b. Reset the values to 0, change the discount to 0.9 and rerun Value Iteration until convergence. What does the policy indicate now? What value does the start state have? (start state value should be 36.9)

4. Now try simulating the agent following the computed policy. Using the "VI Agent" menu, select "Reset state to s0". Then select "Perform 10 actions". The software should show the motion of the agent taking the actions shown in the policy. Since the current setup has 20% noise, you may see the agent deviate from the implied plan. Run this simulation 10 times, observing the agent closely.

   4a. In how many of these simulation runs did the agent ever go off the plan?

   4b. In how many of these simulation runs did the agent arrive in the goal state (at the end of the golden path)?

   4c. For each run in which the agent did not make it to the goal in 10 steps, how many steps away from the goal was it?

   4d. Are there parts of the state space that seemed never to be visited by the agent? If so, where (roughly)?

5. Overall reflections.

   5a. Since it is having a good policy that is most important to the agent, is it essential that the values of the states have converged?

   5b. If the agent were to have to learn the values of states by exploring the space, rather than computing with the Value Iteration algorithm, and if getting accurate values requires re-visiting states a lot, how important would it be that all states be visited a lot?

(OPTIONAL) PART B: Implement Q-Learning (10 points).

Start by renaming the skeleton file YourUWNetID_Q_Learn.py to be your own file, and change the TOH_MDP.py file to import the renamed file (as well as your custom Value Iteration file).

Implement Q-learning to work in two contexts. In one context, a user can "drive" the agent through the problem space by selecting actions through the GUI. After each action is executed by the system, your `handle_transition` function will be called, and you should program it to use the information provided in the call to do your Q-value update.

In the second context, your function `choose_next_action` will be called, and it should perform two things: (1) use the information provided about the last transition to update a Q-value (similar to in the first context), and (2) select an action for the next transition. The action may be an optimal action (based on existing Q-values) or better, an action that makes a controlled compromise between exploration and exploitation.

In order control the compromise, you should implement epsilon-greedy Q-learning. You should provide two alternatives: (a) fixed epsilon, with the value specified by the GUI or the system (including possible autograder), and (b) custom epsilon-greedy learning. In the latter, your program can set an epsilon value and change it during the session, in order to gradually

have the agent focus more on exploitation and less on exploration.

You are encouraged to develop another means of controlling the exploitation/exploration tradeoff. Implementing the use of an exploration function is optional and worth 5 points of extra credit. The GUI provides a means to control whether you are using it or not. However, it is up to you to handle it when it is selected.

One unusual feature of some of the MDPs that we use in studying AI is goal states (or "exit states") and exit actions. In order to make your code consistent with the way we describe these in lecture, adhere to the following rules when implementing the code that chooses an action to return:

- (a) When the current state is a goal state, always return the "Exit" action and never return any of the other six actions.
- (b) When the current state is NOT a goal state, always return one of the six directional actions, and never return the Exit action.

Here is an example of how you can test a state s to find out if it is a goal state:

```
if is_valid_goal_state(s):
  print("It's a goal state; return the Exit action.")
elif s==Terminal_state:
  print("It's not a goal state. But if it's the special Terminal state, return None.")
else:
  print("it's neither a goal nor the Terminal state, so return some ordinary action.")
```

Here is the starter code for this assignment.

What to Turn In.

For Part A, Turn in your Value Iteration Python file, named with the scheme YourUWNetID_VI.py. Also, turn in your slightly modified version of TOH_MDP.py, without changing its name. (The modification this time is that it imports your Value Iterations file. You can leave the import of YourUWNetID_Q_Learn as it is in the starter code for this first submission.)). Finally, be sure to turn in your report file, named using the pattern YourUWNetID_A6_Report.pdf.

For the optional Part B, turn your Q_Learn Python file, named with the scheme YourUWNetID_Q_Learn, and a version of TOH_MDP.py that imports both of your custom modules (your Value Iteration module and your Q_Learn module.)
For Part B, include a section of your report with the heading "Optional Part B:", and describe the work that you did for Part B and the results that you got. Explain how epsilon-greedy Q-learning compared in its fixed-epsilon vs custom-adjustment versions. If you implemented an exploration function, explain what the details of your method were, and what performance changes you got from it.

Do not turn in any of the other starter code files (Vis_TOH_MDP.py, TowersOfHanoi.py, or the actual "YourUWNetID_VI.py" or "YourUWNetID_Q_Learn.py").

Footnotes and Implementation Notes:

*The "golden path" is the shortest path from the initial state to the goal state (the state with all disks on the third peg). You can see the golden path using the starter-code GUI (running TOH_MDP.py from within IDLE), by selecting the MDP Rewards menu and then selecting "show golden path". The "silver path" is the shortest path from the initial state to the special state where the top n-1 disks have been moved to peg 3, but the largest disk remains on peg 1. (The code does not offer a display of the silver path.)

Note that actions are represented in the actions list as strings. You should not need to ever apply actions in your code, because all states for this MDP have already been generated in advance. When you need to get a T(s, a, sp) value, you come up with the three arguments and call the T function directly. The sp states that are relevant to a particular state s can be obtained from STATES_AND_EDGES[s], which is precomputed in the TOH_MDP starter code.

Updates and Corrections:

If necessary, updates and corrections will be posted here and/or mentioned in class or on ED.