Assignment 4: Markov Decision Processes and Computing Utility Values

CSE 473: Introduction to Artificial Intelligence
The University of Washington, Seattle, Autumn 2020

Overview. This assignment not only offers an opportunity to implement the Value Iteration algorithm, but also an opportunity to see how reinforcement learning techniques can be applied to problem solving of the sort we studied earlier with state-space search.

Note that this is an *individual-work* assignment (not collaborative).

The interactions between a problem-solving agent and its problem-domain environment 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 and Part B are both required and worth 50 points each.

NOTE: This assignment requires the use of the Python graphics module known as tkinter, but the version compatible with Python 3.x.

Reports of people who have tried to use Tkinter with other IDEs suggest that 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. It's up to you.

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, 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 Monday, November 16 via Gradescope 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, 3.6, or 3.8, 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, in Part B, the Q-Learning.

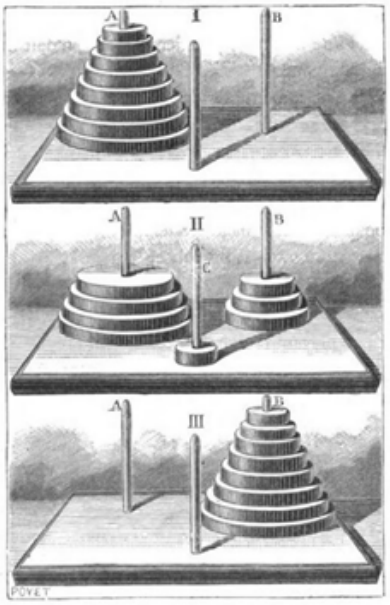

Fig. 36. La tour d'Hanoï.

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 (50 points). There is a starter file called VI.py. Complete the implementation of the functions there:

1. **`one_step_of_VI(S, A, T, R, gamma, Vk)`**
   which will 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 will 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)`**
   which, using the most recently computed Q-state values, will determine the implied policy to maximize expected utility. See the starter code for more details.
4. **`apply_policy(s)`**
   which will 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: "Reset state values (V) and Q values for VI to 0", then "Show state values (V) from VI", and "100 Steps of VI" and finally "Show Policy from VI."

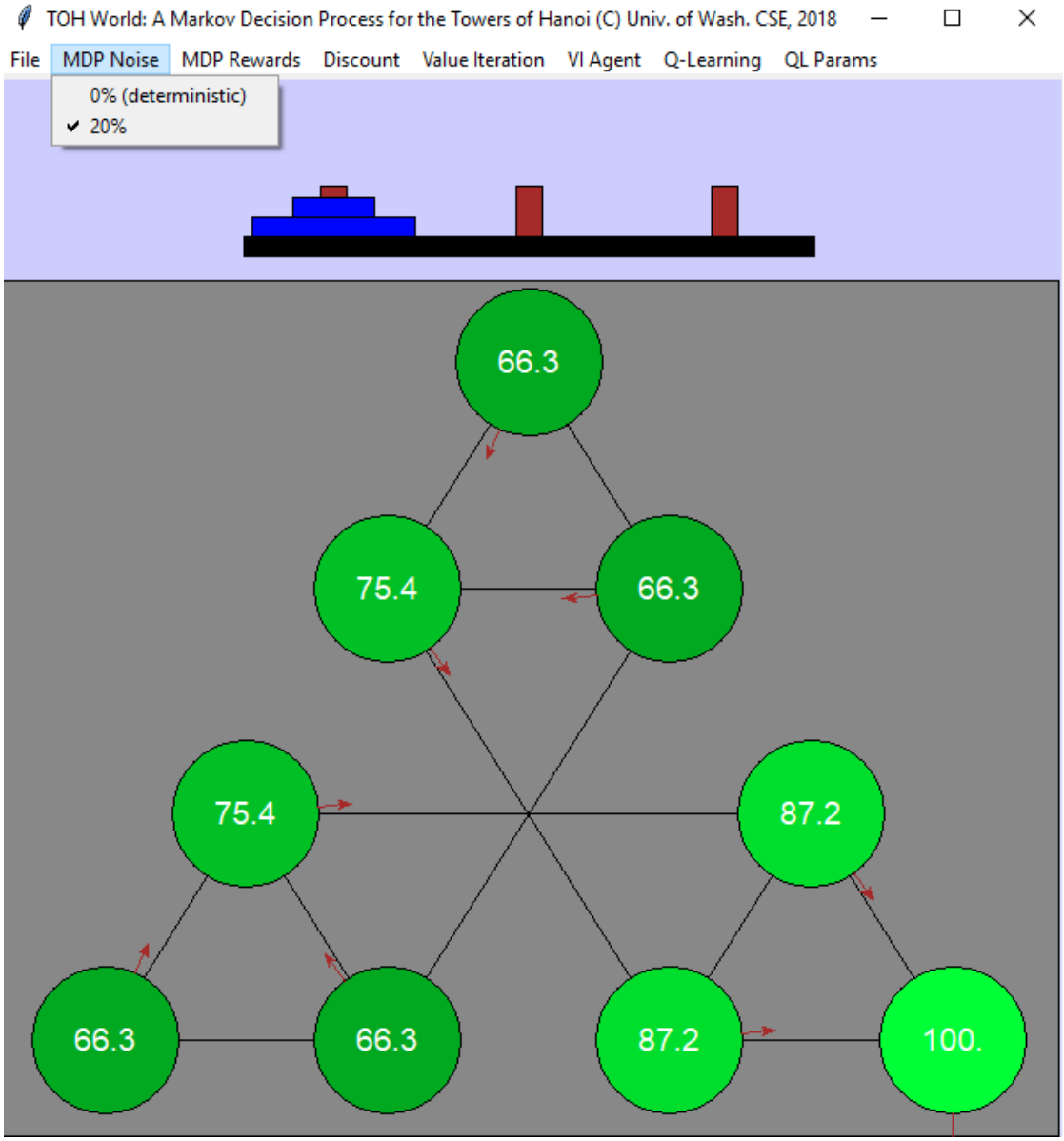

Figure 2. A sample of computed state values. The policy computed from the associated Q values (not shown here) is displayed using red arrows.

Report for Part A.

Create a PDF document that includes the following questions (1a, 1b, etc.) and your answers to them. The document should start with this:

```
Assignment 5 Report for Part A
(your name)
```

Name the file: A5_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?

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

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

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

3. 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.

   3a. In how many of these simulation runs did the agent ever go off the plan?

   3b. In how many of these simulation runs did the agent arrive in the goal state (at the end of the golden path)?

   3c. 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?

   3d. Are there parts of the state space that seemed never to be visited by the agent? If so, where (roughly)?

PART B: Implement Q-Learning (50 points).

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. For auto-grading purposes, you should not add or modify any randomness yourself. Rather, use the provided random generator functions in the skeleton code.

In order to 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 as specified in the starter code. 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. Similarly, you should do the same for the learning rate alpha.

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.")
```

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: "0%(deterministic)"; 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 Q-Learning menu: "Reset state values (V) and Q values for VI to 0", "Reset state to s0", then "Show Q values from QL", "Show policy from QL"; from the QL-Params menu: "Fixed α = 0.2", "Fixed ε = 0.2"; finally, from the Q-Learning menu: "Train for 1000 iterations."

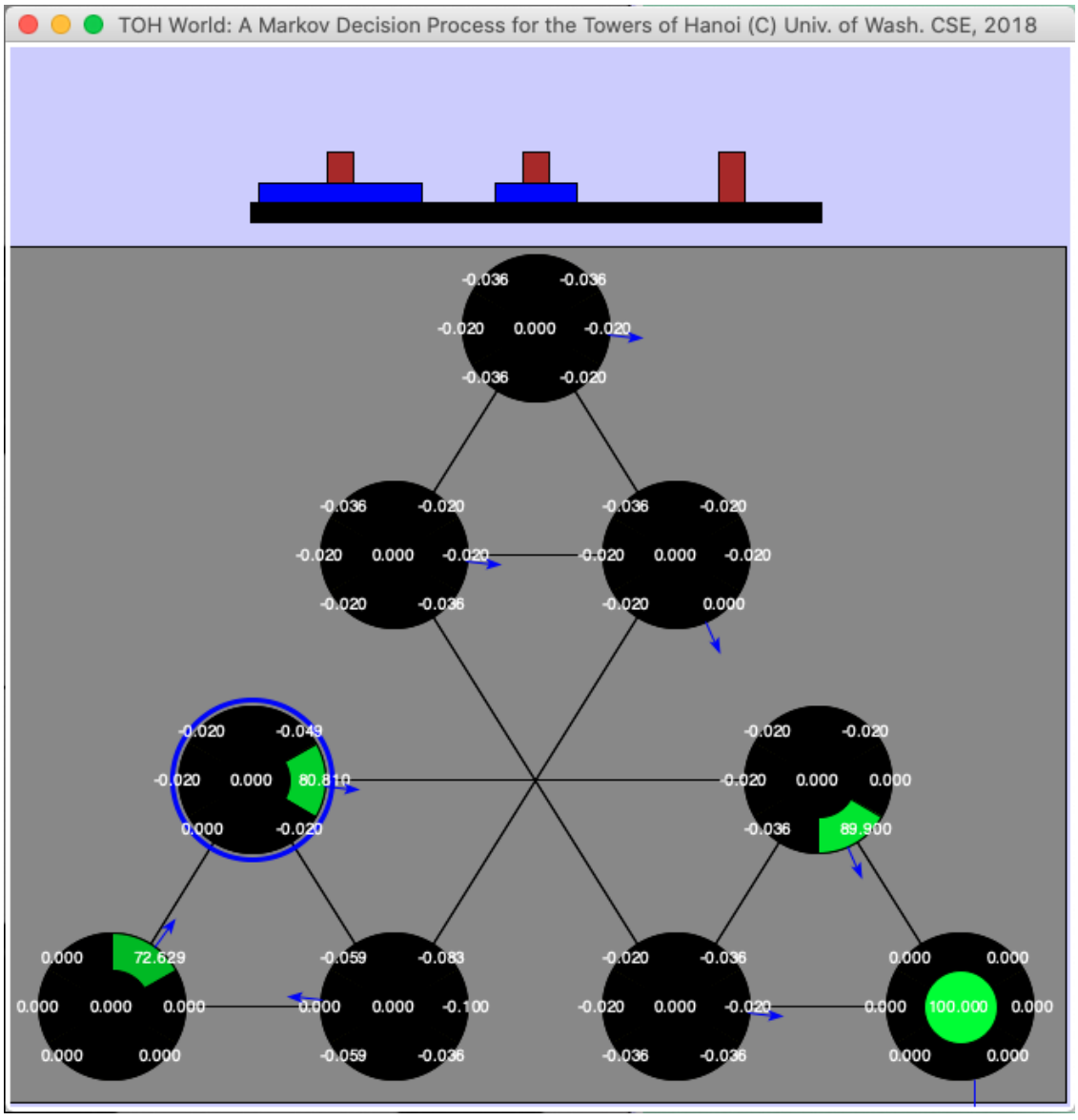

Figure 3. A sample of computed Q values from Q Learning. The policy computed from the associated Q values is displayed using blue arrows. An arrow pointing downward represents the action 'Exit'.

Report for Part B. Answer the following questions in A5_Report.pdf. Note that for some questions, there's no unique answer.

1. Set up the MDP with 2 disks, with 20% noise, and one goal R=100. Find a set of parameters (you can choose any combination of living reward, discount, QL-params), so that after you train for 1000 iterations, the policy shows the golden path.

   4a. Report the set of parameters you chose.

   4b. If you choose an alternative value for each parameter one at a time and train for 1000 iterations each time, how does the optimal policy change? In other words, how sensitive is the trained policy to each parameter (i.e., living reward, discount, α and ε)?

   4c. Now try to set the MDP with 3 disks and still with 20% noise. Try to train for 1000 iterations under a few different settings of parameters. How far are the resulting policies from the golden path? What extra steps would you try in order to improve the training results?

2. Overall reflections.

   5a. In Value Iteration, 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. Comparing Value Iteration to Q learning, with the same number of updates (an update to Q(s, a) both in Value Iteration and Q learning), which one obtains a better policy? Which algorithm is more powerful? What makes the algorithm more powerful than the other?

   Note an update to V(s) in Value Iteration can be written as updates for Q(s, a) for each action a. Then it takes a max over the actions to get V(s). In other words, in Value Iteration, an update to V(s), for a single state s, is equivalent with taking N updates to Q(s, a), where N is the number of valid actions.

   5c. In Q learning, the agent has to learn the values of states by exploring the space, rather than computing with the Value Iteration algorithm. If getting accurate values requires re-visiting states a lot, how important would it be that all states be visited a lot? What parameter would be the best to control the number of re-visitations?

What to Turn In.

For Part A, turn in your Value Iteration Python file, VI.py. For Part B, turn in your Q_Learn Python file. Also turn in your report file, named A5_Report.pdf.

Do not turn in any of the other starter code files (Vis_TOH_MDP.py, TowersOfHanoi.py, or TOH_MDP.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.