CSE 4309 - Assignments - Assignment 8

List of assignment due dates.

The assignment should be submitted via Canvas. Submit a file called assignment8.zip, containing the files:

- answers.pdf, for the output that the programming task asks you to include. Only PDF files will be accepted.
- q_learning.py, containing your Python code for the programming part. Your Python code should run on the versions specified in the syllabus, unless permission is obtained via e-mail from the instructor or the teaching assistant.
- Any other files (if there are any) that are needed in order to document or run your code (for example, additional source code files).

These naming conventions are mandatory, non-adherence to these specifications can incur a penalty of up to 20 points.

Your name and UTA ID number should appear on the top line of both documents.

This assignment has a somewhat different structure than previous assignments. Please read the following guidelines:

- If you do Task 1 correctly, you will get the full 100 points, even if you don't do Task 2 and Task 3. However, Tasks 2 and 3 are easier versions of Task 1, and if you do them you can get 80 points of partial credit even if your Task 1 is not correct. Tasks 2 and 3 are also easier to debug, and almost all their code is useful for Task 1, so you may actually find it quicker to start with Task 2, proceed to Task 3, and then do Task 1 at the end.
- Obviously, the total sum of points for Tasks 1, 2 and 3 will not exceed 100. Tasks 2 and 3 are there to help you develop and test the code for Task 1 more incrementally, and to provide partial credit if Task 1 doesn't work correctly.
- Task 4 simply states that all students will uniformly get some extra credit points based on the percentage of participation in the course evaluation surveys.

Task 1 (100 points, programming)

In this task, you will implement the `AgentModel_Q_Learning` function from the Reinforcement Learning slides.

Function Name and Arguments

File q_learning_main.py contains incomplete code that implements the Q-Learning method for the grid-like environments we have been using as examples in class.

To complete that code, you must create a file called `q_learning.py`, where you implement a Python function called `AgentModel_Q_Learning`. Your function should be invoked as follows:

```
AgentModel_Q_Learning(environment_file, non_terminal_reward, gamma, number_of_moves, Ne)
```

The arguments provide the following information:

- The first argument, `environment_file`, is the path name of a file that describes the environment where the agent moves, and follows the same format as in the `value_iteration` program, **with one exception: the letter I indicating the initial state**.

  More specifically, the environment file will follow the same format as files environment1.txt and environment2.txt. For example, file environment2.txt describes has the following contents:

  ```
  .,.,.,1.0
  .,X,.,-1.0
  I,.,.,.
  ```

  As you see, the environment files are CSV (comma-separated values) files, where:

  - Character 'I' represents the initial state. Your code should assume that the initial state is a non-terminal state.
  - Character '.' represents a non-terminal state that is NOT the initial state.
  - Character 'X' represents a blocked state, that cannot be reached from any other state. You can assume that blocked states have utlity value 0.
  - Numbers represent the rewards of TERMINAL states. So, if the file contains a number at some position, it means that that position is a terminal state, and the number is the reward for reaching that state. These rewards are real numbers, they can have any value.
- The second argument, `non_terminal_reward`, specifies the reward of any state that is non-terminal, as in the `value_iteration` program.
- The third argument, `gamma`, specifies the value of γ that you should use in the utility formulas, as in the `value_iteration` program.
- The fourth argument, `number_of_moves`, specifies how many moves you should process with the AgentModel_Q_Learning function. So, instead of having the main loop run forever, you terminate your program when the `ExecuteAction` function has been called as many times as specified by `number_of_moves`.
- The fifth argument, N_e, is used as discussed in the implementation guidelines, to define the `f` function.

Implementation Guidelines

- As in the value iteration program, for terminal states your model should not allow any action to be performed once you reach those states. Note that the `AgentModel_Q_Learning` pseudocode on the slides does handle this case appropriately, and your implementation should handle this case the same way: terminate the current mission and start a new mission. When starting a new mission, the start state should the initial state specified in the environment file.
- For the η function, use η(N) = 20/(19+N).
- For the f function, use:
  - f(u,n) = 1 if n < N_e, where N_e is the fifth argument to the `q_learning` function.
  - f(u,n) = u otherwise.
- Your solution needs to somehow simulate the `SenseStateAndReward` function, which should be pretty easy. Your solution should also simulate somehow the `ExecuteAction` function, which should implement the state transition model described in pages 9-10 of the MDP slides, with the probabilities that are used in those slides. As described in those slides, bumping to a wall leads to not moving.
- Note that some computations may require values Q[s,a] that have not been instantiated yet. Uninstantiated values in the Q table should be treated as if they are equal to 0.

Output

At the end of your program, you should print out the utility values for all states, as well as the optimal policy, similar to your output for the value iteration algorithm in the previous assignment. More specifically, the output should follow this format:

```
utilities:
%6.3f %6.3f...
...

policy:
%6s %6s...
...
```

In the above output:

- In your utilities printout, each row corresponds to a row in the environment, and you use the %6.3f format specifier for each utility value. For blocked states, just print a utility of 0.
- For the policy printout, again each row corresponds to a row in the environment, and you use the %6s format specifier for each action. To encode each action (or special state) use these characters:
  - For the four actions, use characters `'<'` for `left`, `'>'` for `right`, `'^'` for `up`, and `'v'` for `down`.
  - For blocked states, use character `'x'`.
  - For terminal states, use character `'o'`.

Do NOT print out this output after each iteration. You should only print out this output after the final iteration.

Output for answers.pdf

In your answers.pdf document, you need to provide the complete output for the following invocations of your function:

```
AgentModel_Q_Learning("environment2.txt", -0.01, 0.99, 10000, 1000)
AgentModel_Q_Learning("environment2.txt", -0.04, 0.9, 3000000, 50000)
```

Sample Results

You can see some sample results from my solution at file results_task1.txt

---

Task 2 (Up to 40 points of partial credit, applicable towards Task 1)

In this task, you will implement **passive reinforcement learning**, using a simpler version of `AgentModel_Q_Learning` function from the Reinforcement Learning slides.

Function Name and Arguments

File passive_q_learning_main.py contains incomplete code that implements passive reinforcement learning.

To complete that code, you must create a file called `passive_q_learning.py`, where you implement a Python function called `AgentModel_Q_Learning_Passive`. Your function should be invoked as follows:

```
AgentModel_Q_Learning(environment_file, policy_file, non_terminal_reward, gamma, number_of_moves)
```

For the arguments that are shared with Task 1, please refer to the description of that task. The only different argument is the second argument, `policy_file`, which specifies the policy that should be used. An example of a policy file is policy2_04_09.txt, which is a policy for environment file environment2.txt. For every state, the policy specifies what action should be taken. Actions are specified with `'<'` for `left`, `'>'` for `right`, `'^'` for `up`, and `'v'` for `down`. Character `'x'` indicates a blocked state, and `'o'` indicates a terminal state. Overall, these characters have the same meaning as in the output specifications for Task 1 and for the previous assignment.

The difference between the solution for Task 2 and the solution for Task 1 is the way that the next action is chosen. For Task 1, the next action is chosen as specified in the pseudocode (see the line highlighted in red in slide 60 from the Reinforcement Learning slides). However, for Task 2, that red line should be replaced by appropriate code, that chooses the next action to conform with the given policy.

Output

At the end of your program, you should print out the utility values for all states. **Do not print the policy, since the policy was provided as an input.** More specifically, the output should follow this format:

```
utilities:
%6.3f %6.3f...
...
```

Do NOT print out this output after each iteration. You should only print out this output after the final iteration.

Output for answers.pdf

In your answers.pdf document, please provide the complete output for the following invocations of your function:

```
AgentModel_Q_Learning_Passive("environment2.txt", "policy2_04_09.txt", -0.04, 0.9, 100000)
AgentModel_Q_Learning_Passive("environment2.txt", "policy2_04_09.txt", -0.04, 0.9, 1000000)
```

Sample Results

You can see some sample results from my solution at file results_task2.txt

---

Task 3 (Up to 40 points of partial credit, applicable towards Task 1)

In this task, you will implement **active reinforcement learning** as in Task 1. However, here you will use a simpler version of `AgentModel_Q_Learning` function from the Reinforcement Learning slides. The key difference is that instead of writing code to choose the next action and to determine the result of that action, this information will be provided by an input text file. This way, unlike Task 1 and Task 2, your output for Task 3 should be deterministic and should exactly match my solution.

Function Name and Arguments

File deterministic_q_learning_main.py contains incomplete code that implements this easier version of active reinforcement learning.

To complete that code, you must create a file called `deterministic_q_learning.py`, where you implement a Python function called `AgentModel_Q_Learning_Deterministic`. Your function should be invoked as follows:

```
AgentModel_Q_Deterministic(environment_file, actions_file, non_terminal_reward, gamma, number_of_moves)
```

For the arguments that are shared with Task 1, please refer to the description of that task. The only different argument is the second argument, `actions_file`, which specifies the sequence of actions and consequences.

An example of such a file is actions2a.txt, to be used in conjunction with environment environment2.txt. That actions file contains 10000 lines, specifying 10000 actions and their results. Overall, the actions file is sequence of lines, where the n-th line corresponds to the nth action and has the format:

```
current_row, current_col, action, next_row, next_col.
```

In the above:

- Values `current_row` and `current_col` specify the current state where the agent is when it starts executing the action.
- The action is specified by one character: `'<'` for `left`, `'>'` for `right`, `'^'` for `up`, and `'v'` for `down`.
- Values `next_row` and `next_col` specify the state that is the result of the action.

The main difference Task 1 and Task 2 is the way that the next action is chosen at each step, and the way that the consequences of that action are determined.

- For Task 1, the next action is chosen as specified in the pseudocode (see the line highlighted in red in slide 60 from the Reinforcement Learning slides). However, for Task 3, that red line should be replaced by appropriate code, that chooses the next action to be the one specified by `actions_file`.
- For Tasks 1 and 2, the `ExecuteAction` function determines the outcome of the action in a non-deterministic way, based on the transition model. Here, the `ExecuteAction` function should simply set the result of the action to be the result specified in `actions_file`.

Output

At the end of your program, you should print out the utility values for all states. **Do not print the policy, since the policy was provided as an input.** More specifically, the output should follow this format:

```
utilities:
%6.3f %6.3f...
...
```

Do NOT print out this output after each iteration. You should only print out this output after the final iteration.

Output for answers.pdf

Please follow the same output specifications as in Task 1.

Sample Results

You can see some sample results from my solution at file results_task3.txt

---

Task 4 (Collective Extra Credit, maximum 20 points)

This is a collective extra credit opportunity for the entire class. The goal is to encourage participation in the course evaluations. There are 45 students in the class. Every student will be given the same amount of extra credit points, which will be equal to 1/5 of the percentage of students who submit course evaluations for this class. The amount of extra points will be rounded up to the nearest integer, and added the score of this assignment.

For example, if the class receives 20 course evaluations, every student will receive 9 extra credit points (9 = 20/45 * 100, rounded up). If the class receives 30 evaluations, 14 extra credit points will be given to every student. In the maximum case, if every single student submits a course evaluation, then 20 extra credit points will be given to every student.

Please note that course evaluations are anonymous, there will be no record of who has submitted and who has not. The same number of extra credit will be given uniformly to every student in the class, regardless of whether they have submitted the course evaluation or not.
