CSC 261 **Lab 12: Reinforcement Learning** **Spring 2018**

Assigned: Tuesday 1 May

Due: Friday 11 May, 4:30 pm

Objectives:

- Understand details of **model-free** passive reinforcement learning
- Build an active, **exploratory Q-learning** agent
- Empirically test **agent performance**

Collaboration: This homework assignment will be completed in pairs assigned by the instructor.

Introduction

We conclude our study of learning for Markov decision processes by letting a passive ADP agent forgo a world transition model, directly estimating utilities under a fixed policy. Finally, our most general agent will forgo the world model and learn a policy directly through exploration.

Code

You will need `mdp.c` and `environment.c` from the prior labs. You may obtain the starter code for this assignment on the MathLAN from the directory

```
~weinman/courses/CSC261/code/tdq
```

Here is an overview of the files it contains.

`max.c` Functions for finding the largest value in an array or the index of the largest value in an array over a restricted set of indices (useful for the max/argmax operations in Q-learning).

`td.c` Skeleton for setting up a temporal-difference learning agent.

`qlearn.c` Skeleton for setting up an active Q-learning agent.

Assignment

In addition to those listed below, 12 points will be allotted for general good programming practice.

Problem 1: Temporal-difference learning [16 points]

The file `td.c` contains a setup for running an agent that directly learns utilities (using no world transition model) in a sequential environment for a specified number of iterations. Implement the function `rl_agent_action` according to Passive-TD-Agent of AIMA Figure 21.4 (p. 837).

At the top of the file you will notice several persistent variables that are initialized for you. To assist you, a summary of the conversions from the pseudo code to C statements are given below.

Pseudo-Code	C
current state s' and reward signal r'	parameters `state` and `reward`
s, a, r the previous state, action, and reward	`prevState`, `prevAction`, `prevReward`
$s, a, r \leftarrow$ null	`prevValid = false`
$U[s']$	`utilities[state]`
$N_s[s]$	`state_freq[prevState]`
$\pi[s']$	`policy[state]`
$\alpha(n)$	`double updateWeight(double freq)`

Note that the only environment property available to your agent (via `p_mdp`) is now the `terminal` indication. Use the shell command **`make td`** to build your program with the `Makefile` provided.

Problem 2: Q-Learning [16 points]

The file `qlearn.c` contains a setup for running an agent that directly learns a policy (using no utilities or transition model) in a sequential environment for a specified number of iterations. The Q-Learning-Agent algorithm below gives a pseudo-code implementation appropriate for our environment. In particular, in contrast with the AIMA implementation (Figure 21.8) note that our Q table has no slot for "no available action", *None*. To compensate for this, we set *all* Q values for such states to the reward, and store the maximum Q value accordingly.

Algorithm 1 An exploratory Q-learning agent.

function Q-Learning-Agent(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal, r'
  persistent: Q, a table of action values indexed by state and action, initially zero
    N_{sa}, a table of frequencies for state-action pairs, initially zero
    s, a, r the previous state, action, and reward, initially null
  local variables: Q^*, the maximizing Q-value for the current state

  if Terminal?(s') **then**
    for each action a' **do**
      $Q[s', a'] \leftarrow r'$
    $Q^* \leftarrow r'$
  else
    $Q^* \leftarrow \max_{a' \in A(s')} Q[s', a']$
  if s is not null **then**
    increment $N_{sa}[s, a]$
    $Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma Q^* - Q[s, a])$
  if Terminal?(s') **then**
    $s, a, r \leftarrow$ null
  else
    $s, a, r \leftarrow s', \arg\max_{a' \in A(s')} f(Q[s', a'], N_{sa}[s', a']), r'$
  return a

Implement the `rl_agent_action` function of `qlearn.c`. using Q-Learning-Agent of the algorithm above as a guide. As in prior examples, persistent variables are declared and initialized for you.

Pseudo-Code	C
current state s' and reward signal r'	parameters `state` and `reward`
s, a, r the previous state, action, and reward	`prevState`, `prevAction`, `prevReward`
$s, a, r \leftarrow$ null	`prevValid = false`
$Q[s', a']$	`state_action_value[state][action]`
$N_{sa}[s, a]$	`state_action_freq[prevState][prevAction]`
$f(u, n)$	`double exploration_function( double u, double n )`
$\alpha(n)$	`double updateWeight(double freq)`

Some important notes:

- The `exploration_function` is implemented by using the form on p. 842 with R^+ given by the global variable `bestReward` and N_e given by the global variable `minTries`, both of which are command-line arguments (*`reward`* and *`attempts`*, respectively).
- You can use `max_value` (from `max.h`) to determine Q^*.
- You *cannot* use `arg_max_value` to determine the best f value. Be careful to only evaluate f using the available actions for a state, which is what $a' \in A(s')$ means. (See Exercise B from the MDP lab.)

Use the shell command **`make qlearn`** to build your program with the `Makefile` provided.

Problem 3: Analysis [32 points]

Complete, compiled versions of `td` and `qlearn` are included with the original files copied from the MathLAN; you may use them (with acknowledgment) or your own versions for experimentation. In any case, your report should cite which programs provide your data.

Part A: Temporal Difference Learning

Test Test the passive, model-free TD-learner using the optimal policy learned by policy iteration, e.g.,

```
$ ./policy_iteration gamma epsilon 4x3.mdp > 4x3.policy
$ ./td gamma 4x3.mdp trials < 4x3.policy
```

How many trials does it take before your utilities exactly match nearly all those of Figure 21.1(b)?

Predict Before running the TD-learner on the larger 16×4 world, record how many trials do you expect the agent to require to estimate state utilities accurately? Why?

Experiment Run `td` on the 16×4 grid world using the (presumably) optimal policy learned by policy iteration and the same discount factor and error tolerance you used for the 4×3 grid world. Report the utility of each state (to three significant digits) graphically. How many trials does it take to maximize the number of learned utilities that exactly match those given by value iteration (provided separately)?

Reflect Compare your predictions to experimental outcomes. Speculate on the cause of any discrepancies.

Part B: Q-Learning

Test Test the active, model-free Q-function learner on the 4×3 grid world. What values for the optimistic best *reward* (R^+) and *attempts* (N_e) are required to find the policy closest to optimal at a given number of trials? Why?

Predict Before running the Q-learning agent on the larger 16×4 world, what values of *reward* (R^+), *attempts* (N_e), and trials do you expect your agent to require to learn the best possible policy. Why?

Experiment Run `qlearn` on the 16×4 grid world using the same discount factor you used for the 4×3 grid world. Report the policy graphically. What settings are required for your agent to learn the best policy?

Reflect Compare your predictions to experimental outcomes. Speculate on the cause of any discrepancies.

What to turn in

Your submission should include the following:

- Your completed `td.c` and `qlearn.c` programs
- A single-session transcript `td.txt` of **your** `td` program's compilation (be sure to `make clean` or `rm td` first) and output for *both* grid worlds
- A single-session transcript `qlearn.txt` of **your** `qlearn` program's compilation (be sure to `make clean` or `rm qlearn` first) and output for *both* grid worlds
- A single PDF file `analysis.pdf` containing responses to Problem 3

  C files lacking compile and run transcripts will receive a zero.
  A missing PDF or analysis files in any other format will receive a zero.