COMP 5211 Advanced Artificial Intelligence
Project 3: Learning & Reasoning

Date assigned: Nov 12, 2018

Due time: 11:59PM on Dec 3, 32018

Late submission without prior approval will be penalized by 20% per day

1 Learning

Goal: Implement value iteration and Q-learning.

Preparation: Download and extract `project3.zip`. You will fill in portions of `valueIterationAgents.py` and `qlearningAgents.py` during the project.

Background: MDPs

In problem 1 and 2, you will test your agents on Gridworld. The Gridworld is a maze-like problem. The agent lives in a grid, and walls block the agent's path. The agent's movement is nondeterministic, i.e. actions do not always go as planned, so it is possible that the action North takes the agent West or East. The agent receives rewards each time step, including small "living" reward each step (can be negative) and big rewards come at the end (either good or bad). The goal is to maximize the sum of rewards.

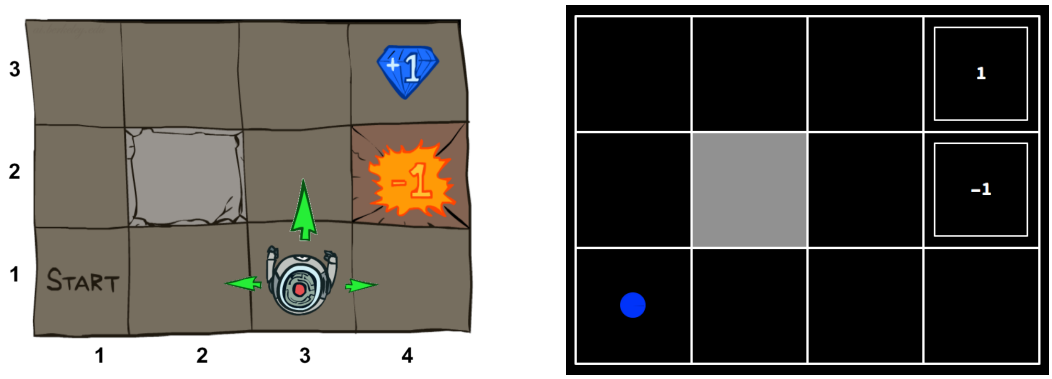
To get started, run Gridworld in manual control mode, which uses the arrow keys:

```
python gridworld.py -m
```

You will see the sample two-exit layout as shown in Figure 1, in which one exit gives positive reward (+1), and the other one is negative (-1). The blue dot is the agent. The default noise is 0.2 (controlled by `-n`), which means that when you press *up*, the agent only actually moves north 80% of the time.

You can control many aspects of the simulation. A full list of options is available by running:

```
python gridworld.py -h
```



(a) Illustration (b) Program GUI

Figure 1: Gridworld

The default agent moves randomly

```
python gridworld.py -g MazeGrid
```

You should see the random agent bounce around the grid until it happens upon an exit.

Note: The Gridworld MDP is such that you first must enter a pre-terminal state (the double boxes shown in the GUI) and then take the special 'exit' action before the episode actually ends (in the true terminal state called `TERMINAL_STATE`, which is not shown in the GUI). If you run an episode manually, your total return may be less than you expected, due to the discount rate (`-d` to change; 0.9 by default).

Look at the console output that accompanies the graphical output (or use `-t` for all text). You will be told about each transition the agent experiences (to turn this off, use `-q`).

As in Pacman, positions are represented by `(x,y)` Cartesian coordinates and any arrays are indexed by `[x][y]`, with `'north'` being the direction of increasing `y`, etc. By default, most transitions will receive a reward of zero, though you can change this with the living reward option (`-r`).

Problem 1: Value Iteration (20 pts)

Write a value iteration agent in `ValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`. Your value iteration agent is an offline planner, not a reinforcement learning agent, and so the relevant training option is the number of iterations of value iteration it should run (option `-i`) in its initial planning phase. `ValueIterationAgent` takes an MDP on construction and runs value iteration for the specified number of iterations before the constructor returns.

Value iteration computes k-step estimates of the optimal values, V_k. In addition to running value iteration, implement the following methods for `ValueIterationAgent` using V_k.

- `computeActionFromValues(state)` computes the best action according to the value function given by `self.values`.

- `computeQValueFromValues(state, action)` returns the Q-value of the (state, action) pair given by the value function given by `self.values`.

These quantities are all displayed in the GUI: values are numbers in squares, Q-values are numbers in square quarters, and policies are arrows out from each square.

Important: Use the "batch" version of value iteration where each vector V_k is computed from a fixed vector V_{k-1}, not the "online" version where one single weight vector is updated in place. This means that when a state's value is updated in iteration k based on the values of its successor states, the successor state values used in the value update computation should be those from iteration k-1 (even if some of the successor states had already been updated in iteration k).

Note: A policy synthesized from values of depth k (which reflect the next k rewards) will actually reflect the next k+1 rewards (i.e. you return π_{k+1}). Similarly, the Q-values will also reflect one more reward than the values (i.e. you return Q_{k+1}).

You should return the synthesized policy π_{k+1}.

Hint: Use the `util.Counter` class in `util.py`, which is a dictionary with a default value of zero. Methods such as `totalCount` should simplify your code. However, be careful with `argMax`: the actual argmax you want may be a key not in the counter!

Note: Make sure to handle the case when a state has no available actions in an MDP (think about what this means for future rewards).

The following command loads your `ValueIterationAgent`, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state (`V(start)`, which you can read off of the GUI) and the empirical resulting average reward (printed after the 10 rounds of execution finish) are quite close.

```
python gridworld.py -a value -i 100 -k 10
```

Hint: On the default BookGrid, running value iteration for 5 iterations should give you the output in Figure 2.

```
python gridworld.py -a value -i 5
```

Grading: Your value iteration agent will be graded on a new grid. We will check your values, Q-values, and policies after fixed numbers of iterations and at convergence (e.g. after 100 iterations).

Problem 2: Q-Learning (30 pts)

Note that your value iteration agent does not actually learn from experience. Rather, it ponders its MDP model to arrive at a complete policy before ever interacting with a real environment. When it does interact with the environment, it simply follows the precomputed policy (e.g. it becomes a reflex agent). This distinction may be subtle in a

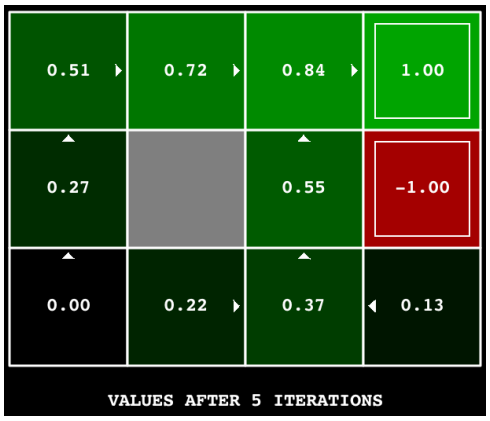


Figure 2: Value iteration

simulated environment like a Gridword, but it's very important in the real world, where the real MDP is not available.

You will now write a Q-learning agent, which does very little on construction, but instead learns by trial and error from interactions with the environment through its `update(state, action, nextState, reward)` method. A stub of a Q-learner is specified in `QLearningAgent` in `qlearningAgents.py`, and you can select it with the option `'-a q'`. For this question, you must implement the `update, computeValueFromQValues, getQValue, and computeActionFromQValues` methods.

Note: For `computeActionFromQValues`, you should break ties randomly for better behavior. The `random.choice()` function will help. In a particular state, actions that your agent *hasn't* seen before still have a Q-value, specifically a Q-value of zero, and if all of the actions that your agent *has* seen before have a negative Q-value, an unseen action may be optimal.

With the Q-learning update in place, you can watch your Q-learner learn under manual control, using the keyboard:

```
python gridworld.py -a q -k 5 -m
```

Recall that `-k` will control the number of episodes your agent gets to learn. Watch how the agent learns about the state it was just in, not the one it moves to, and "leaves learning in its wake." Hint: to help with debugging, you can turn off noise by using the `-noise 0.0` parameter (though this obviously makes Q-learning less interesting). If you manually steer Pacman north and then east along the optimal path for four episodes, you should see the Q-values as shown in Figure 3.

Grading: We will run your Q-learning agent and check that it learns the same Q-values and policy as our reference implementation when each is presented with the same set of examples.

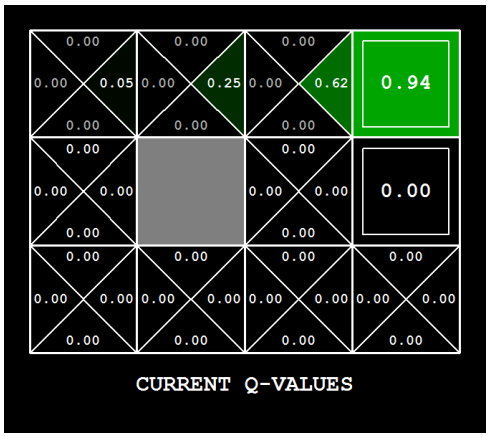


Figure 3: Q-Learning

2 Reasoning

The next two questions are exercises for you to use a mechanical theorem prover to solve problems by reasoning or finding models. The theorem prover you need to use is Z3. The official release is here:
`https://github.com/Z3Prover/z3`
If you use unix, you can install it using pip:
`https://pypi.org/project/z3-solver/#description`
Here is a good description of how to install it by Bob Moore of UT Austin:
`http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/SMT____Z3-INSTALLATION`

Problem 3: The Lady or The Tiger Problem (25 pts)

There are three rooms. Each contains either a lady or a tiger but not both. Furthermore, one room contained a lady and the other two contained tigers. Each of the rooms has a sign, and at most one of the three signs was true. The three signs are:

- Room I: A TIGER IS IN THIS ROOM.
- Room II: A LADY IS IN THIS ROOM.
- Room III: A TIGER IS IN ROOM II.

Which room contains the lady?

Problem 4: The Ranking Problem (25 pts)

Given the following facts:

1. Lisa is not next to Bob in the ranking

2. Jim is ranked immediately ahead of a biology major
3. Bob is ranked immediately ahead of Jim
4. One of the women (Lisa and Mary) is a biology major
5. One of the women is ranked first

What are possible rankings for the four people?

3 Submission

Please submit your solution on Canvas before the due date. You should submit a zip file named as `COMP5211-YourStudentID.zip`, which includes:

- for Problem 1: `valueIterationAgents.py`;
- for Problem 2: `qlearningAgents.py`.
- for Problem 3: `lady.py`, which is a z3 query file for your answer. For example, if your answer is that the lady is in room 2, and you use l2 to denote it, then your z3 query file is to check whether not(l2) is consistent with the KB of this problem. For example, the following python code is a z3 query file to check if q follows from p and $p \supset q$:

```
-------------------------------------------------------------
from z3 import *

p = Bool('p')
q = Bool('q')
s = Solver()
s.add(Implies(p,q))
s.add(p)
# to prove q, add not(q) to see if it causes a contradiction
s.add(not(q))
print(s.check())
-------------------------------------------------------------
```

- for Problem 4: `ranking.py`, which is your z3 query file to find a model from which to read out a ranking. For example, the following is a python code to compute a model of $p \vee q$ and $\neg(p \wedge q)$:

```
-------------------------------------------------------------
from z3 import *

p = Bool('p')
q = Bool('q')
s = Solver()
s.add(or(p,q))
```

```
s.add(not(and(p,q)))
print(s.check())
print(s.model())
------------------------------------------------------------
```

Please *do not* change the other files in the distribution or submit any of our original files other than the files listed above.

Late submissions will be penalized 20% of the score each day (anytime after the due time is considered late by one day).

Please be reminded that this is an individual project. You can discuss with others, but the answers and codes should be written by yourself independently.

Acknowledgment

The first two questions of this project is based on UC Berkeley's AI project[1].

[1] http://ai.berkeley.edu