Semester 2, 2019
COMP3702/7702 ARTIFICIAL INTELLIGENCE
ASSIGNMENT 3: Gridworld & Pacman
Due: 5pm, Saturday 2 November

Acknowledgement: This assignment is based closely on the one created by Dan Klein and John DeNero as part of Berkeley's CS188 course & by Peter Stone as part of the University of Texas at Austin CS343 course.

Introduction

In this project, you will implement Value Iteration and Q-learning. You will test your agents first on Gridworld, then apply them to the game Pacman.

The code for this project contains the following files, which are available in the course Gitlab: https://gitlab.com/3702-2019/assignment-3-support-code

Files you will edit and submit

- `valueIterationAgents.py`
  - A value iteration agent for solving known MDPs.
- `qlearningAgents.py`
  - Q-learning agents for Gridworld and Pacman.
- `analysis.py`
  - e project.

Files you shou

- `mdp.py`
  - Defines methods on general M
- `learningAgents.py`
  - Defines the base classes `Value` `QLearningAgent`, which your agents will extend.
- `util.py`
  - Utilities, including `util.Counter`, which is particularly useful for Q-learners.
- `gridworld.py`
  - The Gridworld implementation.
- `featureExtractors.py`
  - Classes for extracting features on (state, action) pairs. Used for the approximate Q-learning agent (in qlearningAgents.py).

Files you can ignore

- `environment.py`
  - Abstract class for general reinforcement learning environments. Used by `gridworld.py`.
- `graphicsGridworldDisplay.py`
  - Gridworld graphical display.
- `graphicsUtils.py`
  - Graphics utilities.
- `textGridworldDisplay.pyicsUtils.py`
  - Plug-in for the Gridworld text interface.

Note:

- This assignment consists of two parts: Programming and Report.
- You can do this assignment in a group of at most 3 students. This means you can also do the assignment individually.
- For those who choose to work in a group:
    - Please register your group name at the following link before 5pm on Wednesday, 16 October 2019:

https://docs.google.com/spreadsheets/d/1h_0rmdr2ABQJZZzLg529hTHXABPAUdyWG8whiKslkPM/edit#gid=0

- All students in the group must be enrolled in the same course code, i.e., all COMP3702 students or all COMP7702 students.
- All group members are expected to work in both programming and report.

Submission Instructions:

- You will fill in portions of `valueIterationAgents.py`, `qlearningAgents.py`, and `analysis.py` in this assignment. You should submit only these source files. Please don't change any others.
- The report should be in .pdf format and named a3-[courseCode]-[ID].pdf.
    - If you work individually, ID is your student number.
    - If you work in a group, ID is the student number of all group members separated by a dash. For instance, if you work in a group of two, and the student number is 12345 and 45678, then [ID] should be replaced with 12345-45678
- The rep /run your program should [ID]
- The fold 3-[courseCode]-[ID].zip, and the zip file should be sub re **5pm on Saturday, 2 November 2019**.
- If you work in a group, only 1 group m e submission.
- Your code must meet the specifications pt will be made to alter your code, add/remove or rename any files in your submission. Any code that does not meet the correct specification will receive a mark of 0.
- No attempt will be made to download/install any libraries that are not already on the lab computers.
- If you are not using python, you must provide a USAGE file for compiling your code. Your code must be able to compile from a Windows command terminal, you may not open any IDE to compile your code. You must implement the command line arguments listed below.
- There will not be any demos for this assignment.
- Your submission will be hand-marked by tutors, your final grade will be up to the tutors discretion.

MDPs

An MDP describes an environment with observable states and stochastic actions. To experience this for yourself, run Gridworld in manual control mode, and use the arrow keys to move the agent:

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

You will see the two-exit layout from class. The blue dot is the agent. Note that when you perform an action, by default, the agent only actually moves in the intended direction 80% of the time, while 10% of the time it will go off in either of the orthogonal directions.

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

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

The default agent moves randomly. You should see the random agent bounce around the grid until it happens upon an exit:

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

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 sh t this version of Gridworld give states. If you run an episode ma pected, due to the discount rate (`-d`

Look at the console output that accompanies of use `-t` for all text). You will be told about each transitio s (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`).

An MDP can be 'solved' using value iteration.

Question 1. [6 Marks] 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, 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.

- `getValue(state)` returns the value of a state.
- `getPolicy(state)` returns the best action according to computed values.
- `getQValue(state, action)` returns the Q-value of the (state, action) pair.

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.

Note: 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.

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 may assume that 100 iterations is enough for convergence in the questions below.

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 d ations should give you this o

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


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

Question 2. [1 Mark] On `BridgeGrid` with the default discount of 0.9 and the default noise of 0.2, the optimal policy does not cross the bridge. Change only

ONE of the discount and noise parameters so that the optimal policy causes the agent to attempt to cross the bridge. Put your answer in `question2()` of `analysis.py`. (Noise refers to how often an agent ends up in an unintended successor state when they perform an action.) The default corresponds to:

```
python gridworld.py -a value -i 100 -g BridgeGrid --
discount 0.9 --noise 0.2
```


Figure. DiscountGrid layout. This grid has two positive payoff: a close exit with payoff +1 and a di 10. The bottom row of the grid consists of terminal states with negative payoff (shown in red); each state in this "cliff" region has payoff -10. The starting state is the yellow square. We distinguish between two types of paths: (1) paths that "risk the cliff" and travel near the bottom row of the grid; these paths are shorter but risk earning a large negative payoff, and are represented by the red arrow in the figure. (2) paths that "avoid the cliff" and travel along the top edge of the grid. These paths are longer but are less likely to incur huge negative payoffs. These paths are represented by the green arrow in the figure.

Question 3. [5 marks] On the `DiscountGrid` (shown in the figure above), give an assignment of parameter values for discount, noise, and livingReward which produce the following optimal policy types or state that the policy is impossible by returning the string `'NOT POSSIBLE'`. The default corresponds to:

```
python gridworld.py -a value -i 100 -g DiscountGrid --
discount 0.9 --noise 0.2 --livingReward 0.0
```

a. Prefer the close exit (+1), risking the cliff (-10)
b. Prefer the close exit (+1), but avoiding the cliff (-10)
c. Prefer the distant exit (+10), risking the cliff (-10)

d. Prefer the distant exit (+10), avoiding the cliff (-10)
e. Avoid both exits (also avoiding the cliff)

`question3a()` through `question3e()` should each return a 3-item tuple of (discount, noise, living reward) in `analysis.py`.

Note: You can check your policies in the GUI. For example, using a correct answer to 3(a), the arrow in (0,1) should point east, the arrow in (1,1) should also point east, and the arrow in (2,1) should point north.

Q-learning

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

Question 4 (5 marks) 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 environme

`te, reward)`

method. A stu

`qlearningAge` `q`. For this question, you `update getValue getQValue`, and `getPolicy` methods.

Note: For `getPolicy`, you should break ties r aviour. 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: 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!

Question 5. [2 marks] Complete your q-learning agent by implementing epsilon-greedy action selection in `getAction`, meaning it chooses random actions epsilon of the time, and follows its current best q-values otherwise.

```
python gridworld.py -a q -k 100
```

Your final Q-values should resemble those of your value iteration agent, especially along well-travelled paths. However, your average returns will be lower than the Q-values predict because of the random actions and the initial learning phase.

You can choose an element from a list uniformly at random by calling the `random.choice` function. You can simulate a binary variable with probability `p` of success by using `util.flipCoin(p)`, which returns `True` with probability `p` and `False` with probability `1-p`.

Question 6. [1 mark] First, train a completely random Q-learner with the default learning rate on the noiseless BridgeGrid for 50 episodes and observe whether it finds the optimal policy.

```
python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1
```


Now try the sa psilon and a
learning rate f t the optimal
policy will be l should
return EITHER R the string `'NOT POSSIBLE'` if there is none. Epsilon is controll by `-l`.

Approximate Q-learning and Stat

Question 7. [1 mark] Time to play some Pacman! Pacman will play games in two phases. In the first phase, ***training***, Pacman will begin to learn about the values of positions and actions. Because it takes a very long time to learn accurate Q-values even for tiny grids, Pacman's training games run in quiet mode by default, with no GUI (or console) display. Once Pacman's training is complete, he will enter ***testing*** mode. When testing, Pacman's `self.epsilon` and `self.alpha` will set to 0.0, effectively stopping Q-learning and disabling exploration, in order to allow Pacman to exploit his learned policy. Test games are shown in the GUI by default. Without any code changes you should be able to run q-learning Pacman for very tiny grids as follows:

```
python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l
smallGrid
```

Note that `PacmanQAgent` is already defined for you in terms of the `QLearningAgent` you've already written. `PacmanQAgent` is only different in that it has default learning parameters that are more effective for the Pacman problem (`epsilon=0.05, alpha=0.2, gamma=0.8`). You will receive full credit for this

question if the command above works without exceptions and your agent wins at least 80% of the last 10 runs.

Hint: If your `QLearningAgent` works for `gridworld.py` but does not seem to be learning a good policy for Pacman on `smallGrid`, it may be because your `getAction` and/or `getPolicy` methods do not in some cases properly consider unseen actions. In particular, because unseen actions have by definition a Q-value of zero, if all of the actions that ***have*** been seen have negative Q-values, an unseen action may be optimal.

Note: If you want to experiment with learning parameters, you can use the option `-a`, for example `-a epsilon=0.1, alpha=0.3, gamma=0.7`. These values will then be accessible as `self.epsilon, self.gamma` and `self.alpha` inside the agent.

Note: While a total of 2010 games will be played, the first 2000 games will not be displayed because of the option `-x 2000`, which designates the first 2000 games for training (no output). Thus, you will only see Pacman play the last 10 of these games. The number of training games is also passed to your agent as the option `numTraining`.

Note: If you want to watch 10 training games to see what's going on, use the command:


```
python p                    smallGrid -a
numTraini
```


During training, you will see output every 10 about how Pacman is faring. Epsilon is positive during tr l play poorly even after having learned a good policy: this is because he occasionally makes a random exploratory move into a ghost. As a benchmark, it should take about 1,000 games before Pacman's rewards for a 100-episode segment becomes positive, reflecting that he's started winning more than losing. By the end of training, it should remain positive and be fairly high (between 100 and 350).

Make sure you understand what is happening here: the MDP state is the ***exact*** board configuration facing Pacman, with the now complex transitions describing an entire ply of change to that state. The intermediate game configurations in which Pacman has moved but the ghosts have not replied are ***not*** MDP states but are bundled into the transitions.

Once Pacman is done training, he should win very reliably in test games (at least 90% of the time), since now he is exploiting his learned policy.

However, you'll find that training the same agent on the seemingly simple `mediumGrid` may not work well. For example, in some implementations, Pacman's average training rewards remain negative throughout training. At test time, he plays badly, probably losing all of his test games. Training will also take a long time, despite its ineffectiveness.

Pacman fails to win on larger layouts because each board configuration is a separate state with separate q-values. He has no way to generalize that running into a ghost is bad for all positions. Obviously, this approach will not scale.

COMP7702 Extension:

Question 8. [3 marks] Implement an approximate Q-learning agent that learns weights for features of states, where many states might share the same features. Write your implementation in `ApproximateQAgent` class in `qlearningAgents.py`, which is a subclass of `PacmanQAgent`.

Note: Approximate Q-learning assumes the existence of a feature function f(s,a) over state and action pairs, which yields a vector $f_1(s,a)$ .. $f_i(s,a)$ .. $f_n(s,a)$ of feature values. We provide feature functions for you in `featureExtractors.py`. Feature vectors are `util.Counter` (like a dictionary) objects containing the non-zero pairs of features and values; all omitted features have value zero.

The approximate Q-function takes the following form

$$Q(s,a) = \sum_{i}^{n} f_i(s,a) w_i$$

where each weight w_i is associated with a particular feature $f_i(s,a)$. In your code, you should im g features (which the feat wi update your weight vectors

$$w_i \leftarrow w_i + \alpha[correc$$
$$correction = (R(s,a) +$$

Note that the correction term is the same as in normal Q-learning.

By default, `ApproximateQAgent` uses the `IdentityExtractor`, which assigns a single feature to every `(state,action)` pair. With this feature extractor, your approximate Q-learning agent should work identically to `PacmanQAgent`. You can test this with the following command:

```
python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l
smallGrid
```

Important: `ApproximateQAgent` is a subclass of `QLearningAgent`, and it therefore shares several methods like `getAction`. Make sure that your methods in `QLearningAgent` call `getQValue` instead of accessing Q-values directly, so that when you override `getQValue` in your approximate agent, the new approximate q-values are used to compute actions.

Once you're confident that your approximate learner works correctly with the identity features, run your approximate Q-learning agent with our custom feature extractor, which can learn to win with ease:

```
python pacman.py -p ApproximateQAgent -a
extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid
```

Even much larger layouts should be no problem for your `ApproximateQAgent`. (***warning***: this may take a few minutes to train)

```
python pacman.py -p ApproximateQAgent -a
extractor=SimpleExtractor -x 50 -n 60 -l mediumClassic
```

If you have no errors, your approximate Q-learning agent should win almost every time with these simple features, even with only 50 training games.

Congratulations! You have a learning Pacman agent!

The details of the grading scheme are as follows:

Grading for the Programming component

COMP3702:

- COMP3702 students should complete Questions 1-7 **[21 marks]**

COMP7702:

- COMP7702 students should complete ALL Questions i.e. 1-8 **[24 marks]**

Grading for the Report component [10 marks]

In the report pl estions in the coding section

In addition, ple

- Q1. Define the MDP components (states, ac n, reward function) in the context of this t the Bellman Equation for Markov Decision Processes.
- Q2 & 3. Explain how you came up with the discount, noise and living reward values.
- Q4. Describe how Q-learning works. Is this a model-based or model-free approach? Is it a passive or active approach?
- Q5. Describe the epsilon-greedy algorithm and how it is used to balance exploration and exploitation. What alternative methods could have been used?
- Q6. Describe how you came up with the epsilon and learning rate values.
- [COMP7702] Q8. Describe your state abstraction.

All questions should be backed up by data from your experiments and discuss how each section influences the results.

Please format the report with each question under its own heading.