CS461 Artificial Intelligence
Homework 3

Spring 2023

Due: April 23, 2023, 23:59

Instructions

- You will submit your Python codes with.py format to Moodle. Only modify the specified parts (highlighted by `*** YOUR CODE HERE ***` comments) of the provided files, you will lose the whole grade otherwise. There are other parts of the codes you can (but may not need to) change, which are explicitly mentioned in the comments.
- You will submit your codes individually or in groups of two. Please also include the name and the student ID of each member in a text file named `members.txt`.
- You will submit only four files: `valueIterationAgents.py`, `qlearningAgents.py`, `analysis.py`, and `members.txt`. Any other file will be ignored.
- You will need `Python 2.7` installed on your computer to run the environment. Your implementation must not change the package/version requirements. Please note that you might face errors running the scripts if you work on other versions of Python, and you will not be able to run it on `Python 3`. Please do not try to edit the files to make it compatible with `Python 3`. It's recommended to use the Anaconda distribution platform to easily install multiple independent versions of python.
- Only one member should upload a .zip file including the mentioned python files.
- The codes will be checked for plagiarism using online tools which are difficult to fool. This check will include publicly available implementations for this homework. Submit your own work.
- Your codes will also be evaluated in terms of efficiency. Make sure that you do not have unnecessary loops and obvious inefficient calculations in your code.
- We follow no extension policy. However, you will lose 25% of the grade per day of late submission, up to 2 days.
- The files have been tested before being uploaded to Moodle. However, if you still have problems using and running the codes, you can contact the course TAs: Barış Bilgin Şenol (bilgin.senol@bilkent.edu.tr) and Navid Ghamari (navid.ghamari@bilkent.edu.tr).

0 Python Environment Setup

This section describes the Python environment that will be used during the grading of your homework. You do **not** have to follow the same setup as long as you meet the requirements listed in the instructions and get proper grades. However, it is recommended to follow the same setup to avoid inconsistencies that you may encounter while working on the assignment.

The Python environment will be managed by `conda`. If you have not used it before, take this as a chance to get used to that good practice.

1. Download and install Anaconda [1]. Miniconda is also fine if you prefer a minimal installation [2].
2. Search for the `Anaconda Prompt` on your computer. It will launch a command line interface with `base` environment already activated.
3. Create a new `Python 2.7` environment by entering the following command on the Anaconda Prompt. This will create a new `conda` environment named `cs461-hw3` and install `Python 2.7` along with some base packages. Refer to [3] for further information on Conda basics and managing environments.

```
conda create --name cs461-hw3 python=2.7
```

4. Confirm the prompt "Proceed ([y]/n)?" by pressing enter.
5. Enter the following command to the Anaconda Prompt. This will activate the environment you just created. You should see the environment name `cs461-hw3` in parentheses at the beginning of the line.

```
conda activate cs461-hw3
```

6. Enter the following command to verify the Python version.

```
python --version
```

7. You should see an output similar to the one below:

```
>> Python 2.7.16 :: Intel Corporation
```

8. That is it! Run the necessary checks shown under each question in the assignment within this environment. Also, don't forget to select your interpreter as this environment in your code editor of choice.

1 MDP and Reinforcement Learning

Until this point, we have considered the Pac-Man problem as a search problem in which the agent would use search and adversarial search methods to find the best path to the given objective point(s). Another relatively advanced approach to this problem is using Markov Decision Processes (MDP) and Reinforcement Learning (RL). Unlike heuristic approaches, RL agents learn from the experiences and the associated reward/penalty. In this assignment [4], you will implement methods and algorithms related to these approaches.

Question 1 [24 Points]

Value Iteration. Your first task is to implement an offline value iteration agent. Note that this agent is not an RL agent, and it should only estimate the optimal values of the MDP using which the agent will make decisions. The value iteration starts with an initial set of states and then updates the values iteratively in k iterations. You will complete the `ValueIterationAgent` class in the `valueIterationAgents.py` file.

Grading. Your code will be graded by an autograder, but on a different grid. You can still test your implementation on the default grid by running the following command:

```
python autograder.py -q q1
```

GUI. To see how your function updates the values in each iteration, you can run the following command. You can specify a number as the number of iterations. Your implementation will be checked for a limited number of iterations and also at the converged version of the values.

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

Question 2 [4 Points]

Bridge Crossing Analysis. In the second part, you will analyze a specific case. We have a bridge on the two sides of which are a low reward and a high reward terminal. The agent starts from the low reward terminal and the initial policy with a discount value of 0.9 and the noise of 0.2 doesn't allow the agent to cross the bridge and receive the reward. You are asked to change only one of the discount or noise values so that the agent actually crosses the bridge. You can edit the values in the `question2` function in the `analysis.py` file.

Grading. Your code will be graded by an autograder. It's expected that the agent cross the bridge with the new values (only one should change). You can simply run the autograder to see if the new value works:

```
python autograder.py -q q2
```

Question 3 [20 Points]

Policies. In Figure 1, you can see an MDP with a yellow initial state and several states with the associated positive and negative scores. There are two types of paths that the agent can choose. The red one is shorter but with a higher chance of severe penalty, and the green path is longer and less risky. The distant exit (target) is in green with a 10 point reward, the near exit with a single point reward, and the cliff line with 10 negative points are colored in red. In this question, you will choose settings of the discount, noise, and living reward parameters for this MDP to produce optimal policies of several types. The different types of policies are as follows:

(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 and the cliff (so an episode should never terminate)

You will implement these policies in the functions from `question3a()` to `question3e()` in the `analysis.py` file.

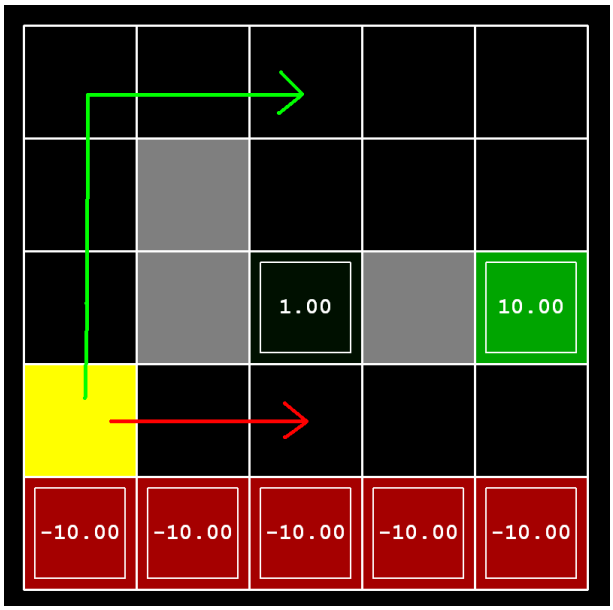

Figure 1: Discount Grid [4]

Grading. The returned policy will be checked for each case in the evaluation of your implementation. You can test your implementation by running the command below:

```
python autograder.py -q q3
```

Question 4 [20 Points]

Q-Learning. The value iteration function you implemented never interacts with the environment and simply follows the precomputed policy. We were able to implement the value iteration function based on the available MDP. However, the MDP is not usually available in the realistic scenarios, and therefore the only way for the agent to improve is to learn while taking actions and interacting with the environment. In this question, you will implement a Q-Learning agent by completing the `update`, `computeValueFromQValues`, `getQValue`, and `computeActionFromQValues` functions in the `qlearningAgents.py` file.

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 has not seen before still have a Q-value, specifically a Q-value of zero, and if all the actions that your agent has seen before have a negative Q-value, an unseen action may be optimal.

Important: Make sure that in your `computeValueFromQValues` and `computeActionFromQValues` functions, you only access Q-values by calling `getQValue`. This abstraction will be useful for Question 8 when you override `getQValue` to use features of state-action pairs rather than state-action pairs directly.

Grading. Your code will be graded by an autograder. You can evaluate your implementation of the Q-Learning agent by running the following command:

```
python autograder.py -q q4
```

Question 5 [12 Points]

Epsilon Greedy. Exploration and exploitation is an important balance in a reinforcement learning problem, since a random action might be the optimal action. Now, you should complete the Q-Learning agent by completing the `getAction` function in the `qlearningAgents.py` file, which allows random actions in which the agent can explore the environment as opposed to following the known path. Note that your random actions should be truly random and not suboptimal random cases.

Grading. Your code will be graded using the autograder. Note that your implementation should work on different layouts, and to test this we will run the autograder on different cases. You can test your implementation with the default test case using the following command:

```
python autograder.py -q q5
```

Question 6 [4 Points]

Bridge Crossing Revisited. First, train a completely random Q-learner with the default learning rate on the noiseless `BridgeGrid` layout 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 same experiment with an epsilon of $\epsilon = 0$. Is there an epsilon and a learning rate for which it is extremely likely that the optimal policy will be learned after 50 iterations? `question6()` in `analysis.py` should return **either** a 2-item tuple of `(epsilon, learning rate)` **or** the string `'NOT POSSIBLE'` if there is none. Epsilon and learning rate are controlled by `-e` and, `-l` respectively.

Note: Your response should not depend on the exact tie-breaking mechanism used to choose actions. This means your answer should be correct even if, for example, the entire bridge grid world is rotated 90 degrees.

Grading. Your code will be graded using the autograder. Note that your implementation should work on different layouts, and to test this we will run the autograder on different cases. You can test your implementation with the default test case:

```
python autograder.py -q q6
```

Question 7 [4 Points]

Q-Learning for Pac-Man. Eventually, we want to use the implemented Q-Learning agent to have a learning version of the previous Pac-Man agent. Pac-Man with Q-Learning has two phases. In the first phase, the agent learns about the values of the positions and actions. The training phase is done without GUI since the training takes too long. In the second phase, which is the test phase, Pacman's `self.epsilon` and `self.alpha` will be set to 0.0, effectively stopping Q-learning and disabling exploration, in order to allow Pac-Man to exploit its learned policy. You can run the Pac-Man agent to test the learned policy by running the following command which will run a total of 2010 games, 2000 of which are training games.

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

Grading. Your code again will be graded using the autograder. The autograder will run 100 test games after the 2000 training games. You can test your implementation running the following command:

```
python autograder.py -q q7
```

Note: Your Q-Learning agent might work well in question 4 while failing in Pac-Man agent. This might be because your `getAction` and `computeActionFromQValues` functions do not work properly. You may check the reference web page [4] for further details.

Question 8 [12 Points]

Approximate Q-Learning. Finally, implement an approximate Q-Learning agent which assigns values to each of the n features of a state. This approximate agent might assign the same value for different states, as different states can potentially share the same features. You should complete the `ApproximateQAgent` function in the `qlearningAgents.py` file.

The approximate Q-Learning agent will have a Q-function as below. The function will assign values for every feature and finally assign a score to the state.

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a) \tag{1}$$

You can run the following command to train and test your implemented agent in two phases similar to the previous question.

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

Grading. Your code will be graded using the autograder. The autograder will compare the learned Q-values and feature weights with those of the reference implementation. You can test your implementation running the following command:

```
python autograder.py -q q8
```

2 Grading

Although your homework will be graded using an autograder with the mentioned details, your implementations will also be checked. In any case, only the correct implementations will be accepted.

References

[1] Anaconda web page. `https://www.anaconda.com/`. Accessed: 2023-04-04.

[2] Miniconda documentation. `https://docs.conda.io/en/latest/miniconda.html`. Accessed: 2023-04-04.

[3] Conda cheat sheet. `https://docs.conda.io/projects/conda/en/4.6.0/_downloads/52a95608c49671267e40c689e0bc00ca/conda-cheatsheet.pdf`. Accessed: 2023-04-04.

[4] Berkeley project web page. `https://inst.eecs.berkeley.edu/~cs188/sp23/projects/proj4/`. Accessed: 2023-04-04.