Introduction to AI - A.Y. 2024/2025

Project 2: Markov Decision Processes / Reinforcement Learning

Fall '24

Abstract

This project belongs to the projects series from the Introduction to AI Course by Berkeley University. Further information can be found at the link `http://ai.berkeley.edu`.
Modules and data are stored in the `Project2BMDPs.zip` folder from the course page.
At the end of this project, students complete the course topic on Markov Decision Processes. Specifically, they will be able to implement and analyze MDPs algorithms.

1 Introduction

1.1 Problem & Purpose

Students implement value iteration algorithm and understand the meaning of its hyperparameters in this project. The project is subdivided into *Tasks* that students have to refer to. To submit their work, students should refer to Section 5.

1.2 Project Folder & Modules

The code for this project consists of several Python files, some of which have to be read and understood to complete the assignment and some of which can be ignored by the students. All the modules belong to the `Project2BMDPs.zip` folder from the course webpage.

Files to be edited: [1]

- `valueIterationAgents.py`: value iteration agent used to solve known MDPs.
- `analysis.py`: a file to put your answer to questions given in the project.

[1] *DO NOT* change the other files

Files to be read but *NOT* edited:

- `mdp.py`: here, the methods on general MDPs are defined.
- `learningAgents.py`: this file defines the base class *ValueEstimationAgent*, that your agent will extend.
- `util.py`: utilities module.
- `gridworld.py`: the Gridworld implementation.
- `featureExtractors.py`: the module's classes are used to extract features on (state, action) pairs.

Other project files are briefly described in Appendix A.

Running the project: After downloading the code, unzipping it, and changing the directory, one can get started by running Gridworld in manual control mode:

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

This is the basic command students could refer to to understand whether they correctly set the working folder. The agent is displayed as a blue dot. Note that when pressing *up*, the agent only actually moves north 80% of the time.
Once this one doesn't generate any issues, another command to be used to ensure the right functioning of the starting code is the following:

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

You should see the random agent bounce around the grid until it happens upon an exit.
The module `gridworld.py` supports other commands too. A full list of options is available by running:

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

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 together with 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`).

2 Task 1 (6 pts): Value Iteration

Students have to write a value iteration agent in *ValueIterationAgent*, which has been partially specified in `valueIterationAgents.py`.

Note 1: This *ValueIterationAgent* is an offline planner, not a reinforcement learning agent, so the important training option is the number of iterations of value iteration it should run (option `-i`) in its initial planning phase.

Note 2: *ValueIterationAgent* takes an MDP on construction and runs value iteration for this specific number of iterations, k, before the constructor returns. Value iteartion computes k-step estimates of the optimal values, V_k.

Note 3: In addition to running value iteration, you should 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.

Note 4: Students should 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 5: 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}.
To help yourself, 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 6: Make sure to handle the case when a state has no available actions in an MDP (think about what this means for future rewards).

Code check: To test the implementation, students should run the autograder:

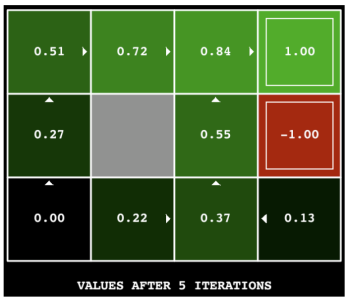
```
python autograder.py -q q1
```

The following commands loads the *ValueIterationAgent*, which will compute a policy and execute it 10 times. By pressing a key, one cycles through values, Q-values, and the simulation. One should find that the value of the start state (*V(start)*, that one can read on 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
```

On the default BookGrid, running value iteration for 5 iterations
`python gridworld.py -a value -i 5`
should give the output in Figure 1.

Figure 1:



3 Task 2 (1 pts): Bridge Crossing Analysis

BrideGrid (Figure 2) is a grid world map with a low-reward terminal state and a high-reward terminal state separated by a narrow 'bridge', on either side of which is a chasm of high negative reward. The agent starts near the low-reward state. With the default discounting factor of 0.9 and the default noise of 0.2 (noise refers to how often an agent ends up in an unintended successor state when they perform an action), the optimal policy does not cross the bridge.

Students have to change only ONE between the discount and noise parameters so that the optimal policy causes the agent to attempt to cross the bridge.

The answer should be put in *question2()* of `analysis.py`. The default corresponds to:

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

Figure 2:

Code check: To check your answer, run the autograder:

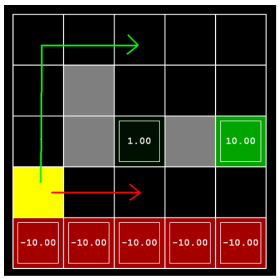
```
python autograder.py -q q2
```

4 Task 3 (5 pts): Policies

Consider the *DiscountGrid* layout (Figure 3). This grid has two terminal states with positive payoff (in the middle row), a close exit with payoff +1 and a distant one with payoff +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:

- 'risk the cliff' paths, that is, those paths near the bottom of the grid; these paths are shorter path risk earning a large negative payoff (they are represented by the red arrow in the Figure).
- 'avoid the cliff' paths, i.e., those paths along the top edge of the grid; they are longer but are less likely to incur huge negative payoffs (green arrow in the Figure).

Figure 3:



Students have to choose settings of the discount, noise, and living reward parameters for this MDP to produce optimal policies of several different types.
The optimal policy types you should produce are:

a. Prefer to 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).

Note 1: Your setting of the parameter values for each part should have the property that, if your agent followed its optimal policy without being subject to any noise, it would exhibit the given behavior. If a particular behavior is not achieved for any setting of the parameters, you should return the string 'NOT POSSIBLE'.

Note 2: 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.

Code check: To check the answer, run the autograder:

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

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

5 Submission

To submit their work, students must upload their folder on the course page on Microsoft Teams.

The purpose of module `autograder.py` is to briefly evaluate students' work. Using the autograder is helpful for both students and instructors. In particular, there is no limit to the number of times it can be used.

A Further Files: Info

- `environment.py`: abstract class for general reinforcement learning environments.
- `graphicsGridworldDisplay.py`: gridworld graphical display.
- `graphicsUtils.py`: graphics utilities.
- `textGridworldDisplay.py`: ASCII graphics for the Gridworld text interface.
- `crawler.py`: the crawler code and test harness.
- `graphicsCrawlerDisplay.py`: GUI for the crawler robot.
- `autograder.py`: project autograder.
- `testParser.py`: parses autograder test and solution files.
- `testClasses.py`: general autograding test classes.
- `reinforcementTestClasses.py`: project 2B specific autograding test classes.
- `test_cases/`: directory containing the test cases for each question.