CSE 190 - Q-Learning Exploration

Author: Le Lu (A99048469), Huajun Zhang (A11031299)

Introduction

In this project, we are interested in implementing Q-learning as an extension to PA3. Similar to MDP, we want to find an optimal policy for each grid in a given map. However, this time probabilistic models are not known and have to be learned. This can be a common case where robot has no idea how accurate its movements are and Q-learning ensures the optimization of the final policy list even when robot takes undesired movements in some iterations.

Assumptions

We made several assumptions in our experiments and they are reflected in **configuration.json**, **robotmover.py** and **learning.py**. Similar to PA3, robot knows its possible moves (move_list), map information (map_size, start, goal, walls, pits) and reward for special occasions (moving steps, hitting wall, reaching goal, falling in pit). It also knows the discount factor and learning rate.

As being said, this time the robot has no idea how accurately it can move. To simulate its movement, we designed a class **RobotMover** in **learning.py** that has a function **take_action()** which takes in a current state and an action, and returns robot's landing state based off the probabilistic models. Note that probabilistic models are ONLY perceived in this class; the robot, however, is not exposed to any information of probabilistic models.

Algorithm

1/ Load map and initialize all grids as a **PGrid** object. **PGrid** entails reward for all four directions and numbers of movements robot has taken for each direction.
2/ Initialize Q value for each direction in each grid. (i.e. Each grid has four directions and each direction has an individual Q value.)
3/ Loop through all grids in the map. For each non-wall/pit/goal grid, choose one direction to explore.

- The direction is chosen based off a **F value** which is the sum of Q value and L value of that direction in that grid. ** L value - the incentive of moving in this direction*
- *L value = (a random fixed constant C/ the total number of movements robot take on that direction in that grid)*
- Direction with the greatest F value among four directions represents the best direction to explore in the current iteration: it is guaranteed to be either
    - a direction with a large positive reward (so taking more movements on that direction, the reward will likely increase again) OR
    - a direction that has not been explored before (L value will be very huge.)

4/ With the current state, selected direction (action), call **take_action()** to get the robot's landing state. Afterwards, update new Q value for that direction in current grid.

- **take_action()** is recognized as a library call - as being said, it applies the underlying probabilistic models to determine where the robot lands from an input state through an input action.
- To update Q value of that direction in that grid, apply

*Q(S, a, S') = (1-alpha)*Q(S) + alpha * (R(S, a, S') + Q(S') * d)* , where

- Q(S, a, S'): new Q value of state S when robot lands on state S' from state S through action a
- Q(S): current Q value of state S
- R(S, a, S'): reward received when robot lands on state S' from state S through action a
- alpha: learning rate
- d: discount factor

5/ Repeat 3-4. Run for a huge number of iterations (e.g. 1000 iterations). *similar to PA3 MDP.
6/ Output results and compare results with MDP.

Expected Output

We estimated that Q-learning output should be very similar to MDP output. However, we understood that it can be "unstable" - that is, running learning algorithm for multiple times may result in different outputs because the learning rate **alpha** and **constant C** (used to compute L-value) are all variables affecting the result. Also, **take_action()** uses **random()** which does not produce a "truly random" number.

Results & Comparison

It turned out that Q-learning result mostly matched MDP output. We compared the output of both algorithm with different configurations **(configuration.json and configuration_2_10_10_1.json)**: although under some cases, Q-learning output did not match MDP outputs exactly, it was acceptable since the policy computed always avoids pits.

Also, as we expected, Q-learning output varies with different **alpha** and **C**. By increasing the learning rate from 0.001 to 0.02, we observed fluctuation in the result. We also noticed that the number of iterations plays a role in determining the final policy list after playing around with the number. A general rule can be observed as: the more iterations there are and the finer learning rate is, the more closely Q-learning performs as MDP.

Reflection - which is better?

It is hard to say which algorithm is definitely better than the other. Q-learning algorithm is an alternative of computing the policy list with less information with small sacrifice of accuracy.

Besides the situation where the robot has no clue of probabilistic models, Q-learning can be used as a "safe blanket" when computing the policy list: although sometimes robot knows its movement probabilities, they might appear deceptively. (or, what if the robot lies?)
Q-learning result may not be the most optimal but it is guaranteed to be not so far away from the truth. It is very useful and we believe that its accuracy can always be improved with some other deeper learning filters.

Links

Github: https://github.com/raphaellu/Q-learning-Exploration
Youtube: https://www.youtube.com/watch?v=Jc0C8loJt3Q