Reinforcement Learning Tech Task
AID

Harsh Sharma
harshsha@andrew.cmu.edu

12 January, 2020

1. **PROBLEM OVERVIEW**
   The problem is basically a maze-solving problem, which has obstacles and each action has a probability of slipping based on the height difference. The goal is to reach the goal location from the start location.

   Note: Code execution instructions are provided in the last section of the documentation.

2. **DESIGN CHOICES AND RATIONALE**
   The problem is an MDP with a discrete state and action space with a sparse reward function. The goal is to find a policy using reinforcement learning. Since both states and actions were discrete and finite- my initial thoughts were to use Value Iteration and solve the problem. I thought of iterating over all the states a certain number of times or till the point where the Bellman error reaches a certain threshold. This should have ultimately converged to a feasible policy. Theoretically this should have worked, but the practical implementation realities were a different ball-game altogether.

3. **VALUE ITERATION**
   I coded the Value Iteration solution for the problem. When I ran the code I realised that the code took a long while to complete just one pass over the states. The map size was 2048X2048. This implies in one pass I was updating the value function of almost 4 million states. Also for each state 8 actions existed, thus further increasing the time requirements. Thus, the time complexity of Value Iteration O(| SXA |) per pass is a real pain in practical implementation of Value Iteration.
   Here I realized that not all state's Value function needs to be updated. An obstacle (water for our case) would always remain an obstacle. So we can simply set the value function of the cells which are an obstacle to be -inf and not update them later. This way we can reduce the number of cells to be updated in each pass.
   Even after doing this I felt that each pass was still taking a lot of time and the algorithm should hence be improved.

   Assumptions: The robot can't keep standing at it's present cell. Each action tries to lead it to a different state. It may however slip and end up in the same cell.

   Note: See code execution instructions mentioned in the last section of the documentation.

4. **Q-LEARNING**
   I felt that Q-learning would also suffer from a similar issue. It would also be highly memory intensive as each state has 8 actions. The Q value table size would have been 2048X2048X8. Hence the idea of implementing Q-learning was shunned.

   Assumptions: The robot can't keep standing at it's present cell. Each action tries to lead it to a different state. It may however slip and end up in the same cell.

5. **REAL-TIME DYNAMIC PROGRAMMING (RTDP)- FINAL SOLUTION**
During the implementation of Value Iteration I realized that there were a lot of states which are getting updated redundantly if the problem is a single-query problem of going to the goal location from a start location. So I thought of using RTDP which would help converge to a feasible policy faster. RTDP is a very popular alternative to Value Iteration and is in fact more efficient than it. It does not compute value function for all the states but in fact focuses on the relevant states. So basically I greedily pick a roll-out based on the maximum value function for each of the successor states. This is followed by backing up all the states in the roll-out. This process is repeated till all the states in the current greedy policy have Bellman errors within a threshold value.
There is a possibility that I may not ever reach the goal from the given state. To avoid that case I limit the roll-out path length based on a higher factor of the Euclidean distance between the start and the goal locations*. I found out that a factor of 7-10 worked good enough. This also helped in faster convergence. Another thing that I tried was backing up the states in the reverse order. So I stored all the resulting states in the roll-out and updated them together later in reverse order. This also did improve the convergence rate of the algorithm. However, often times this led to a greedy path. This is because once a path to the goal is found, all the states in this path get a high value function and hence this path is selected almost always. So the algorithm quickly converges to this greedy policy, without exploring for a new one.

However reaching the goal in the roll-outs is essential for convergence in each of the case. If the algorithm doesn't reach the goal state in any of it's roll-outs it would not be able to yield a policy. In that case, either the solution doesn't exist or it might exist but was not obtained for the current maximum number of episodes and the factor of the Euclidean Distance. We can play around with these two parameters to obtain the solution if it exists.

*More on this can be found in section 7.2

6. **SUCCESS CASES**

6.1. **Example 1:**
Start_state is: [524 1346] Goal State: [334 1373]

Convergence of value function has been achieved after: 17639 episodes

Total training time is 34499.0355348587 seconds

Path to goal found. Goal reached successfully. Total reward is: 623.1156569650368

For complete path to the goal from the start state refer to: ***successful_case_1.txt***

6.2. **Example 2:**
Start_state is: [1914 1547] Goal State: [1600 1519]

Convergence of value function has been achieved after: 12614 episodes

Total training time is 8700.555232048035 seconds

Path to goal found. Goal reached successfully. Total reward is: 479.46385334942704

For complete path to the goal from the start state refer to: ***successful_case_2.txt***

6.3. **Example 3:**
Start_state is: [694 409] Goal State: [1158 432]

Total training time is 25921.48033308983 seconds

Convergence NOT achieved after: 25000 episodes

Path to goal achieved successfully! Total reward is: -147.58282784479889

For the best feasible solution obtained refer to: ***successful_case_3.txt***

Note: This is the best roll-out which reached the goal during the training. The algorithm didn't however converge.

7. **FAILURE CASES**

Based on my approach I can list down a couple of cases in which the approach might fail to yield a solution.

7.1. **Failure to reach the goal**

If the RTDP roll-outs never reach the goal even once during all it's iterations, none of the states have a positive value function because because the reward function has all negative rewards apart from reaching the goal. So it is highly likely that the resulting policy might not yield a path to the goal state and hence in this case the algorithm doesn't guarantee completeness of the solution.

Recovering from failure: There are two possible causes of this. The first one is that the MAX_EPISODES counter was less than required. This can happen in a case where the goal is really far from the start state and more episodes are required to alter the value functions and hence lead to convergence. Theoretically this situation would never arise if we could iterate infinite times over the states. However practically speaking it is an issue.

The second case is that a path from the start location to the goal location doesn't exist. In that case increase the MAX_EPISODES counter won't help. A path simply doesn't exist!

7.2. **Insufficient Maximum episode length**

The code uses a parameter by the name of MAXIMUM_EPISODE_LENGTH to limit the length of the roll-out. This is done to avoid the case where a path from a start location to the goal location is not possible. If that happens the roll-out can extend to cover all states in each episode which will be highly inefficient. The MAXIMUM_EPISODE_LENGTH factor in the final version of the code is set to 7 times the Euclidean distance between the start and the goal location.

However, using this variable adds a case of failure ie. Maybe a path to the goal exists however is longer than the MAXIMUM_EPISODE_LENGTH I provided. In that case the algorithm would never be able to reach the goal and hence completeness of the solution is compromised.

Recovering from failure: Try and increase this factor if in none of the instances the agent observed the goal. If now it did, that means this was the case of failure.

8. **FUTURE WORK**

8.1. **Adding a component of exploration**

I can try and explore the case of RTDP with reverse backup with an attempt to explore (like decaying-epsilon greedy approach). This could help the circumvent the issue of fixing on a greedy policy.

8.2. **Value Iteration with more computation**

Increasing the iteration cycles on the value iteration algorithm by giving it more time and compute resources. This might lead to convergence and could in turn lead to one of the best and easiest solutions to this problem of finding an optimal policy in this case.

9. **Running instructions**

9.1. **RTDP (Final Solution)**

```
python rtdp.py
```

9.2. **Value Iteration**

```
python vi.py
```