Imperial College London

October 23, 2018

Department of Computing

CO424H – Learning in Autonomous Systems (LIAS) – Dr Aldo Faisal

Lab Assignment A

Lab assignment: Understanding of MDPs

Consider a stair climbing MDP made of 7 states $\mathcal{S} = \{s_1, \ldots, s_7\}$. The top of the states and the bottom of the states are absorbing states s_1, s_7. Reaching the bottom yields you $+100$ reward, reaching the bottom yields you -100 reward. Each step up gives you -10 reward (effort), each step down give you 10 reward. The two possible actions $\mathcal{A} = \{Down, Up\}$ are deterministic. We do not specify a specific γ.

1. Draw the MDP as a graph (by hand)
2. Using below python code as template for specifying an MDP, specify above Stair Climbing MDP (You will have to fix the code).
3. Implement code that takes a deterministic policy (i.e. $\pi(s) = a$) and computes the value function for this policy for $\gamma = \frac{1}{2}$
4. Compute for the "Always Down" policy and the "Always Up" policy the two value functions (code it so that it works for an arbitrary $\gamma \in [0, 1]$.).
5. Plot how varying γ will make an "always up" policy be more rewarding than an "always down" if you start in s_4.
6. Think about a way to find a better deterministic policy (e.g. by choosing actions so as to be greedy on the value function).

```
import numpy as np
import matplotlib.pyplot as plt

#   Basic code to specify an MDP
#   Learning in Autonomous Systems coursework
#   Aldo Faisal (2015), Imperial College London


class StairClimbingMDP(object):
    def __init__(self):
        # States are:  { s1 <-- s2 <=> s3 <=> s4 <=> s5 <=> s6 --> s7 ]
        self.S = 7
        self.state_names = ['s1', 's2', 's3', 's4', 's5', 's6', 's7']

        # Actions are: {L,R} --> {0, 1}
        self.A = 2
        self.action_names = ['L','R']
```

```
        # Matrix indicating absorbing states
        # P  1   2   3   4   5  6  7  G   <-- STATES
        self.absorbing = [1, 0, 0, 0, 0, 0, 1]

        # Load transition
        self.T = self.transition_matrix()

        # Load reward matrix
        self.R = self.reward_matrix()

    # get the transition matrix
    def transition_matrix(self):
        # MODIFY HERE
        #                    1      ...      7 <-- FROM STATE
        TL = np.array([[0,0,0,0,0,0,0],    # 1 TO STATE
                       [0,0,0,0,0,0,0],    # .
                       [0,0,0,0,0,0,0],    # .
                       [0,0,0,0,0,0,0],    #
                       [0,0,0,0,0,0,0],    #
                       [0,0,0,0,0,0,0],    #
                       [0,0,0,0,0,0,0]])   # 7


        # MODIFY HERE
        #                    1      ...      7 <-- FROM STATE
        TR = np.array([[0,0,0,0,0,0,0],    # 1 TO STATE
                       [0,0,0,0,0,0,0],    #
                       [0,0,0,0,0,0,0],    # .
                       [0,0,0,0,0,0,0],    # .
                       [0,0,0,0,0,0,0],    # .
                       [0,0,0,0,0,0,0],    #
                       [0,0,0,0,0,0,0]])   # 7


        return np.dstack([TL,TR]) # transition probabilities for each action


    # the transition subfunction
    def transition_function(prior_state, action, post_state):
        # Reward function (defined locally)
        prob = self.T(post_state,prior_state, action)
        return prob



    # get the reward matrix
    def reward_matrix(self, S = None, A = None):
        # i.e. 11x11 matrix of rewards for being in state s,
        # performing action a and ending in state s'

        if(S == None):
            S = self.S
```

```
    if(A == None):
        A = self.A

    R = np.zeros((S,S,A))

    for i in range(S):
        for j in range(A):
            for k in range(S):
                R[k,i,j] = self.reward_function(i,j,k)

    return R

# the locally defined reward function
def reward_function(self,prior_state, action, post_state):
    # reward function (defined locally)
    # MODIFY HERE
    if ((prior_state == 1) and (action == 0) and (post_state == 0)):
        rew = -1
    elif ((prior_state == 5) and (action == 1) and (post_state == 6)):
        rew = 1
    elif (action == 0):
        rew = 0
    else:
        rew = 0

    return rew
```