Coursework 3
Reinforcement Learning (CO-424H)

Adrian Löwenstein

November 22, 2018

1 Understanding of MDPs

1) The trace generated by my CID (01594572) is $\tau = s_2\ 1\ s_0\ 1\ s_1\ 0\ s_2\ 1\ s_0\ 1\ s_2\ 0$

2) We can create the MDP graph displayed in figure 1.

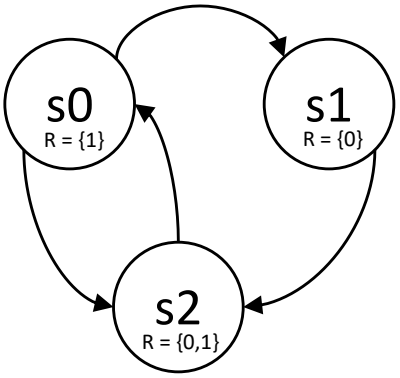


Figure 1: MDP Graph for my CID

a) We are here discussing the transition matrix and the reward function of our process.

The transition matrix that can be deduced from the limited length of our trace is :

$$\begin{bmatrix} 0 & p_{01} & p_{02} \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \tag{1.1}$$

We can see from the matrix that the process is not deterministic and rather stochastic. From state s_0 there is 2 possible output states s_1 and s_2.

We also observe that the reward is not always the same when leaving a state. The first time we leave the state s_2 the reward is 1, as for the last step, the reward for leaving the state s_2 is 0. Therefore we can state the reward function is stochastic in our MDP.

b) From our trace we can compute the value of the state s_0. We observe the rewards after we pass in state s_0 and use a discount factor of $\gamma = 1$. We have then : $V(s_0) = 3$. This is the only way to estimate a value function as we have an undefined stochastic process.

2 Understanding of Grid Worlds

For this part we provide in addition to the following comments, two annexes containing the optimal policy and values for my CID : 01594572, and the full code for obtaining the results of the coursework in the form of a Jupyter Notebook.

1) My CID is 01594572, therefore we have $x = 5$, $y = 7$, $z = 2$. As a result we have that $p = 0.5$ and $\gamma = 0.65$.

2) We compute the optimal value function and the optimal policy by using the Value Iteration Algorithm. I used the class provided in Lab2 of this module. To the basis class, I added my own function `optimal_value_policy()`, which implements the Value Iteration Algorithm. The tolerance used for the convergence of the algorithm is 0.0001. The results can be observed in the figure 2 :

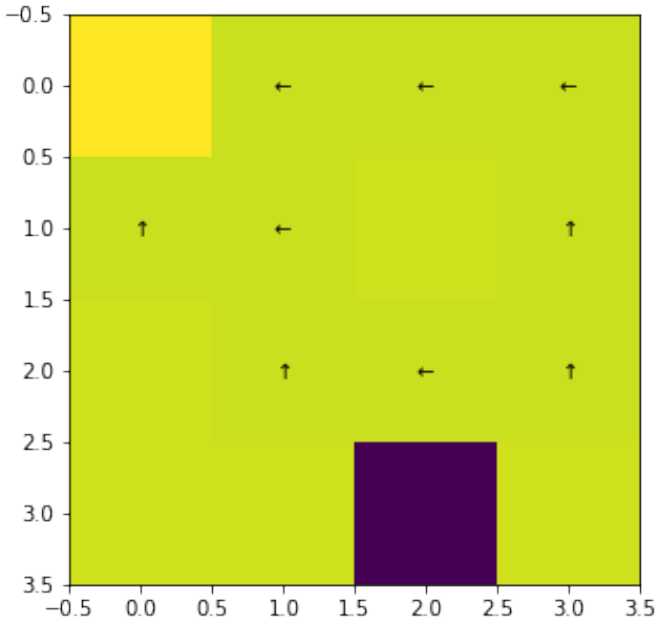


Figure 2: Gridworld - Optimal Policy

The Optimal Value Functions for the each state are written in the following table :

State	Value Function
s_1	0
s_2	5.28
s_3	0.73
s_4	-1.33
s_5	5.91
s_6	1.22
s_7	-2.57
s_8	-3.76
s_9	-21.65
s_{10}	-5.33
s_{11}	0

3) In state $s9$ the optimal action a is "left". The probability of this optimal action in 1 as we are in a deterministic process. But the fact that this action may not lead to the expected state in highly depending on the probability of realisation p.

In our CID case we have $p = 0.5$, to the probability of not leading to the expected state is $p' = (1-p)/3$, as $p > p'$ we will more likely end in the wanted state. If p gets small enough the probability wrong transition (p') gets larger than p. If we change the value of p to be small (such as $p = 0.1$, see figure 3), one can observe that the optimal action is "down". At first sight this should be the worse action, but because of p, this has actually very little chance of happening.

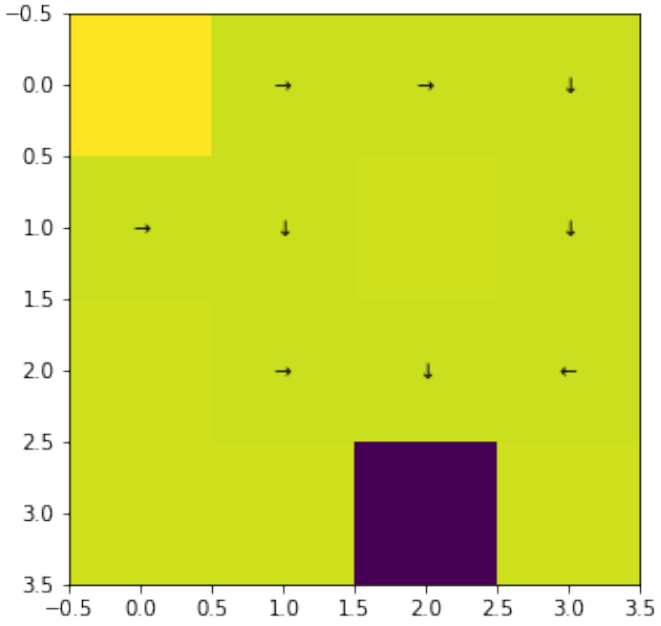


Figure 3: Gridworld - Optimal Policy - p = 0.1

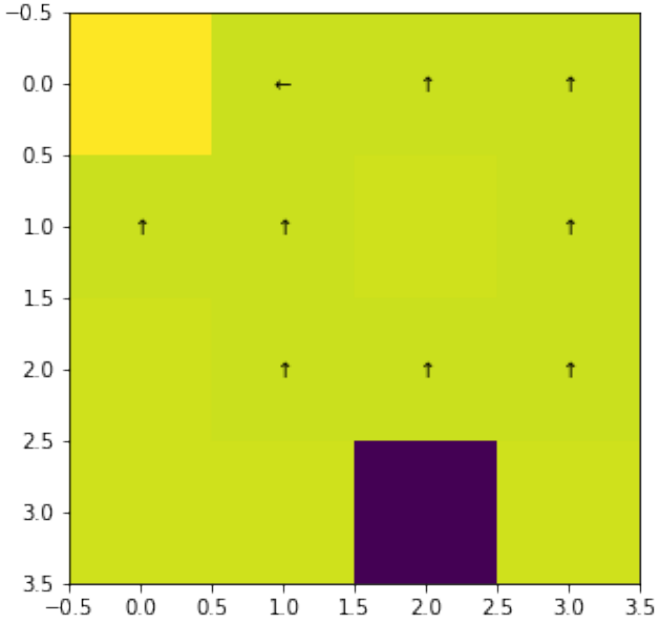


Figure 4: Gridworld - Optimal Policy - $\gamma = 0$

We can also discuss the influence of γ. For a small value of gamma we get a very "short-sighted" algorithm that can't see any long term rewards. If we change the discount factor to $\gamma = 0$ (see figure 4) we see that the optimal action for s_9 is going "up", in the wall. This shows that it don't want to move from its current state, as it only sees negative rewards immediately close to it.

4) With a global point of view for our grid world, we clearly see that influence of p and γ is really huge on the optimal values function and policy. In the case of our CID, the specific values of the parameters lead to specific behavior at certain points. For example at the state s_6, we can imagine that both "up" and "left" could be optimal actions. But as p is not maximal, it is a safer choice to go on the left, in order to avoid the possibility of going to state s_3 from s_2. Also the value of γ has an influence on how it values the far away rewards. The specific values of p and γ have therefore a big influence on the optimal value functions and policies.

Name: Adrian Löwenstein
CID: 01594572
reward state: s1
$p =$ 0.5
$\gamma =$ 0.65

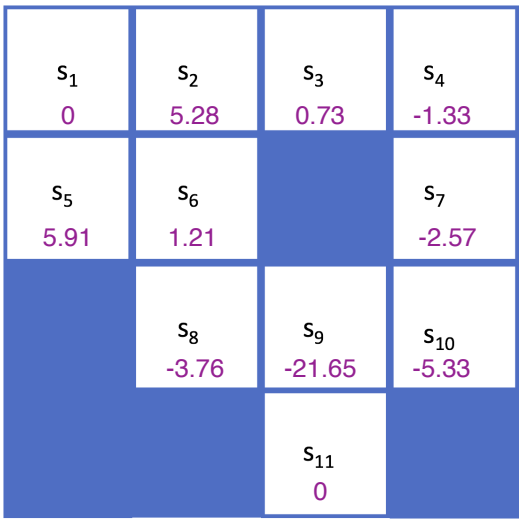


Figure 1: Optimal value function. Values for each state rounded to 2 decimal places.

.

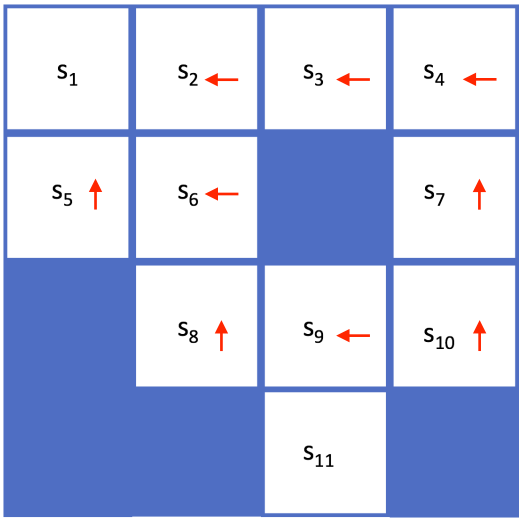


Figure 2: Optimal policy. Arrows indicate optimal action direction for each state (deterministic policy), multiple arrows from one state indicate equiprobable choice between indicated directions (stochastic policy).

.

424H_CW1

November 22, 2018

1 CW1 - Reinforcement Learning

Import usefull libraries.

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (8,5)
```

1.1 Part 1 - Understanding of MDPs

In [3]:
```
CID = "1594572"
#CID = "12345678"
CID = np.array([int(CID[i]) for i in range(len(CID))])
CID
```

Out[3]:
```
array([1, 5, 9, 4, 5, 7, 2])
```

In [4]:
```
s = np.mod(CID + 1 ,3)
s
```

Out[4]:
```
array([2, 0, 1, 2, 0, 2, 0])
```

In [5]:
```
r = np.mod(CID[1:] ,2)
r
```

Out[5]:
```
array([1, 1, 0, 1, 1, 0])
```

In [6]:
```
tau = np.array([ [s[i],r[i]] for i in range(len(r))]).flatten()
tau
```

Out[6]:
```
array([2, 1, 0, 1, 1, 0, 2, 1, 0, 1, 2, 0])
```

In [7]:
```
x = CID[-3]
y = CID[-2]
z = CID[-1]

#Generation of personalized Data :
j = np.mod(z+1,3)+1
print(j)
```

```
        p= 0.25 + 0.5 *x/10.
        print(p)
        gamma = 0.3+0.5* y /10.
        print(gamma)
```

```
1
0.5
0.6499999999999999
```

1.2 Part 2 - Understanding of Grid Worlds

```
In [8]: class GridWorld(object):
            def __init__(self):

                ### Attributes defining the Gridworld #######
                # Shape of the gridworld
                self.shape = (4,4)

                # Locations of the obstacles
                self.obstacle_locs = [(2,0),(3,0),(3,1),(3,1),(3,3),(1,2)]

                # Locations for the absorbing states
                self.absorbing_locs = [(0,0),(3,2)]

                # Rewards for each of the absorbing states
                self.special_rewards = [10, -100 ] #corresponds to each of the absorbing_locs

                # Reward for all the other states
                self.default_reward = -1

                # Starting location
                self.starting_loc = (0,3)

                # Action names
                self.action_names = ['N','E','S','W']

                # Number of actions
                self.action_size = len(self.action_names)


                # Randomizing action results: [1 0 0 0] to no Noise in the action results.
                self.action_randomizing_array = [ p, (1-p)/3, (1-p)/3 , (1-p)/3]

                ############################################
```

```
#### Internal State  ####


# Get attributes defining the world
state_size, T, R, absorbing, locs = self.build_grid_world()

# Number of valid states in the gridworld (there are 11 of them)
self.state_size = state_size

# Transition operator (3D tensor)
self.T = T

# Reward function (3D tensor)
self.R = R

# Absorbing states
self.absorbing = absorbing

# The locations of the valid states
self.locs = locs

# Number of the starting state
self.starting_state = self.loc_to_state(self.starting_loc, locs);

# Locating the initial state
self.initial = np.zeros((1,len(locs)))
self.initial[0,self.starting_state] = 1


# Placing the walls on a bitmap
self.walls = np.zeros(self.shape);
for ob in self.obstacle_locs:
    self.walls[ob]=1

# Placing the absorbers on a grid for illustration
self.absorbers = np.zeros(self.shape)
for ab in self.absorbing_locs:
    self.absorbers[ab] = -1

# Placing the rewarders on a grid for illustration
self.rewarders = np.zeros(self.shape)
for i, rew in enumerate(self.absorbing_locs):
    self.rewarders[rew] = self.special_rewards[i]

#Illustrating the grid world
self.paint_maps()
################################
```

```
####### Getters ###########

def get_transition_matrix(self):
    return self.T

def get_reward_matrix(self):
    return self.R


########################

####### Methods #########
def policy_evaluation(self, policy, threshold, discount):

    # Make sure delta is bigger than the threshold to start with
    delta= 2*threshold

    #Get the reward and transition matrices
    R = self.get_reward_matrix()
    T = self.get_transition_matrix()

    # The value is initialised at 0
    V = np.zeros(policy.shape[0])
    # Make a deep copy of the value array to hold the update during the evaluation
    Vnew = np.copy(V)

    # While the Value has not yet converged do:
    while delta>threshold:
        for state_idx in range(policy.shape[0]):
            # If it is one of the absorbing states, ignore
            if(self.absorbing[0,state_idx]):
                continue

            # Accumulator variable for the Value of a state
            tmpV = 0
            for action_idx in range(policy.shape[1]):
                # Accumulator variable for the State-Action Value
                tmpQ = 0
                for state_idx_prime in range(policy.shape[0]):
                    tmpQ = tmpQ + T[state_idx_prime,state_idx,action_idx] * (R[stat

                tmpV += policy[state_idx,action_idx] * tmpQ

            # Update the value of the state
```

```
                Vnew[state_idx] = tmpV

            # After updating the values of all states, update the delta
            delta =  max(abs(Vnew-V))
            # and save the new value into the old
            V=np.copy(Vnew)

        return V

    def draw_deterministic_policy(self, Policy):
        # Draw a deterministic policy
        # The policy needs to be a np array of 22 values between 0 and 3 with
        # 0 -> N, 1->E, 2->S, 3->W
        plt.figure()

        plt.imshow(self.walls+self.rewarders +self.absorbers)
        #plt.hold('on')
        for state, action in enumerate(Policy):
            if(self.absorbing[0,state]):
                continue
            arrows = [r"$\uparrow$",r"$\rightarrow$", r"$\downarrow$", r"$\leftarrow$"]
            action_arrow = arrows[action]
            location = self.locs[state]
            plt.text(location[1], location[0], action_arrow, ha='center', va='center')

        plt.show()
    ############################


    ############ Internal Helper Functions ######################
    def paint_maps(self):
        plt.figure()
        plt.subplot(1,3,1)
        plt.imshow(self.walls)
        plt.subplot(1,3,2)
        plt.imshow(self.absorbers)
        plt.subplot(1,3,3)
        plt.imshow(self.rewarders)
        plt.show()

    def build_grid_world(self):
        # Get the locations of all the valid states, the neighbours of each state (by
        # and the absorbing states (array of 0's with ones in the absorbing states)
        locations, neighbours, absorbing = self.get_topology()

        # Get the number of states
        S = len(locations)
```

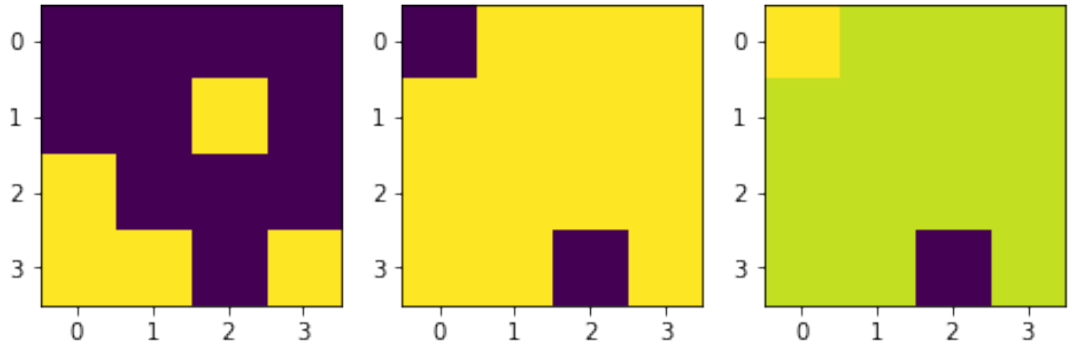
```
        # Initialise the transition matrix
        T = np.zeros((S,S,4))

        for action in range(4):
            for effect in range(4):

                # Randomize the outcome of taking an action
                outcome = (action+effect+1) % 4
                if outcome == 0:
                    outcome = 3
                else:
                    outcome -= 1

                # Fill the transition matrix
                prob = self.action_randomizing_array[effect]
                for prior_state in range(S):
                    post_state = neighbours[prior_state, outcome]
                    post_state = int(post_state)
                    T[post_state,prior_state,action] = T[post_state,prior_state,action]


        # Build the reward matrix
        R = self.default_reward*np.ones((S,S,4))
        for i, sr in enumerate(self.special_rewards):
            post_state = self.loc_to_state(self.absorbing_locs[i],locations)
            R[post_state,:,:]= sr

        return S, T,R,absorbing,locations

    def get_topology(self):
        height = self.shape[0]
        width = self.shape[1]

        index = 1
        locs = []
        neighbour_locs = []

        for i in range(height):
            for j in range(width):
                # Get the locaiton of each state
                loc = (i,j)

                #And append it to the valid state locations if it is a valid state (ie
                if(self.is_location(loc)):
                    locs.append(loc)

                    # Get an array with the neighbours of each state, in terms of loca
                    local_neighbours = [self.get_neighbour(loc,direction) for directio
```

```
                neighbour_locs.append(local_neighbours)

    # translate neighbour lists from locations to states
    num_states = len(locs)
    state_neighbours = np.zeros((num_states,4))

    for state in range(num_states):
        for direction in range(4):
            # Find neighbour location
            nloc = neighbour_locs[state][direction]

            # Turn location into a state number
            nstate = self.loc_to_state(nloc,locs)

            # Insert into neighbour matrix
            state_neighbours[state,direction] = nstate;


    # Translate absorbing locations into absorbing state indices
    absorbing = np.zeros((1,num_states))
    for a in self.absorbing_locs:
        absorbing_state = self.loc_to_state(a,locs)
        absorbing[0,absorbing_state] =1

    return locs, state_neighbours, absorbing




def loc_to_state(self,loc,locs):
    #takes list of locations and gives index corresponding to input loc
    return locs.index(tuple(loc))


def is_location(self, loc):
    # It is a valid location if it is in grid and not obstacle
    if(loc[0]<0 or loc[1]<0 or loc[0]>self.shape[0]-1 or loc[1]>self.shape[1]-1):
        return False
    elif(loc in self.obstacle_locs):
        return False
    else:
         return True

def get_neighbour(self,loc,direction):
    #Find the valid neighbours (ie that are in the grif and not obstacle)
    i = loc[0]
    j = loc[1]
```

```
        nr = (i-1,j)
        ea = (i,j+1)
        so = (i+1,j)
        we = (i,j-1)

        # If the neighbour is a valid location, accept it, otherwise, stay put
        if(direction == 'nr' and self.is_location(nr)):
            return nr
        elif(direction == 'ea' and self.is_location(ea)):
            return ea
        elif(direction == 'so' and self.is_location(so)):
            return so
        elif(direction == 'we' and self.is_location(we)):
            return we
        else:
            #default is to return to the same location
            return loc

    def optimal_value_policy(self, threshold, discount):
        # Computes the Optimal Value and Policy with the Value Itteration Algorithm

        # Make sure delta is bigger than the threshold to start with
        delta= 2*threshold

        #Get the reward and transition matrices
        R = self.get_reward_matrix()
        T = self.get_transition_matrix()

        # The value is initialised at 0
        V = np.zeros(self.state_size)
        # The policy is initialised at 0
        policy = np.zeros((self.state_size,), dtype=np.int)

        # Make a deep copy of the value array to hold the update during the evaluation
        Vnew = np.copy(V)

        # While the Value has not yet converged do:
        while delta>threshold:
            for state_idx in range(self.state_size):
                # If it is one of the absorbing states, ignore
                if(self.absorbing[0,state_idx]):
                    continue

                # List variable for the Value of a state
                tmpQ_list = []
                for action_idx in range(self.action_size):
                    # Accumulator variable for the State-Action Value
```

```
                        tmpQ = 0
                        for state_idx_prime in range(self.state_size):
                            tmpQ = tmpQ +
                                T[state_idx_prime,state_idx,action_idx] *
                                (R[state_idx_prime,state_idx, action_idx] +
                                discount * V[state_idx_prime])

                            tmpQ_list.append(tmpQ)

                        # Update the value of the state
                         # Choosing the largest valued Policy (greedy)

                        policy[state_idx] = int(np.argmax(tmpQ_list))
                        Vnew[state_idx] = tmpQ_list[policy[state_idx]]

                    # After updating the values of all states, update the delta
                    delta =  max(abs(Vnew-V))
                    # and save the new value into the old
                    V=np.copy(Vnew)

                return policy, V

        ############################################
```

In [9]:
```
myWorld = GridWorld()
```



In [10]:
```
optPolicy, optV = myWorld.optimal_value_policy(0.0001,gamma)
```

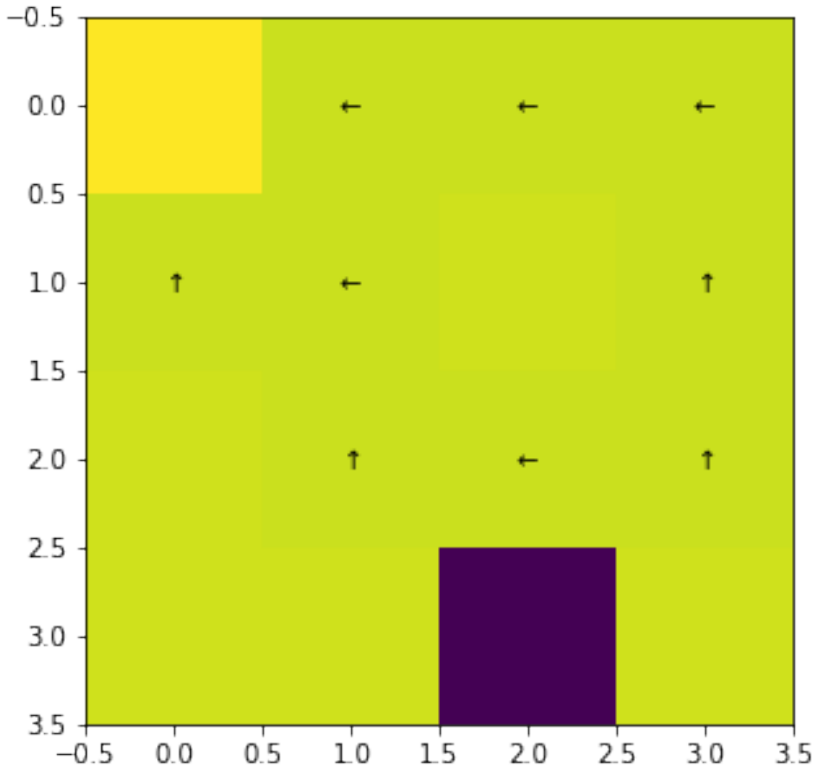
In [11]:
```
optPolicy
```

Out[11]:
```
array([0, 3, 3, 3, 0, 3, 0, 0, 3, 0, 0])
```

In [12]:
```
optV
```

```
Out[12]: array([  0.        ,   5.28366338,   0.73191542,  -1.32774645,
                  5.91316542,   1.21829509,  -2.56530113,  -3.76482201,
                -21.64655263,  -5.33465824,   0.        ])
```

```
In [13]: myWorld.draw_deterministic_policy(optPolicy)
```




```
In [ ]:
```