Imperial College London

COURSEWORK #1

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Reinforcement Learning

Author:
Alexandre Allani
aa4719@ac.ic.uk (CID: 01797836)

Date: November 1, 2019

1 Question 1: Understanding MDPs

1.1

I have $CID = 01797836$, this gives me a trace:

$$\boxed{\tau = s_0\ 1\ s_0\ 1\ s_2\ 1\ s_0\ 1\ s_1\ 0\ s_2\ 1\ s_2\ 0}$$

1.2

1.2.1

With my specific trace we can infer graph on **Figure 1**

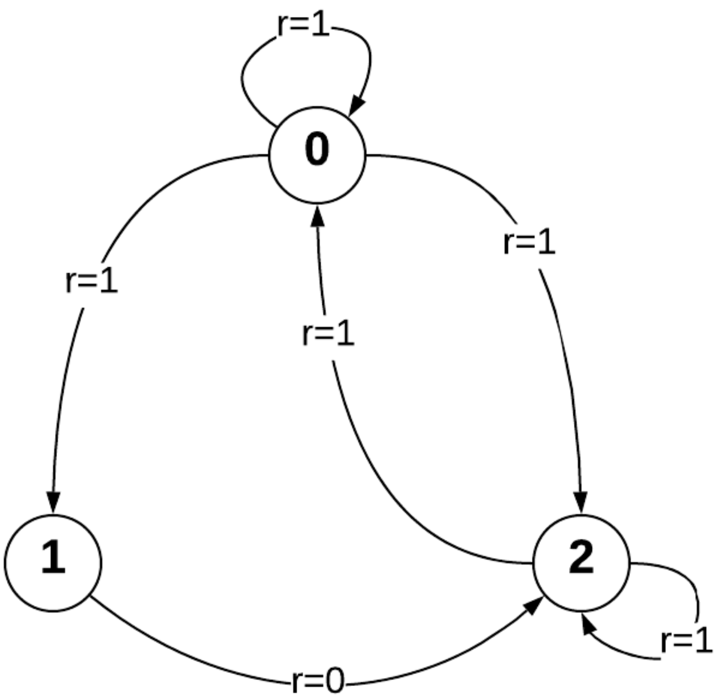

Figure 1: MDP graph

- We can suppose given the trace, that there is no terminal state, or that the trace of this episode is not finished, since we have a reward at the end of the trace. Moreover, the transitions are not deterministic, hence the transition matrix is stochastic. For instance in state s_0, we can go either to s_0, s_1 or s_1. Based on this assumption, the transition matrix could be like this :

$$T = \begin{bmatrix} p_{0,0} & X & p_{0,2} \\ p_{1,0} & X & X \\ p_{2,0} & p_{2,1} & p_{2,2} \end{bmatrix} \text{ where } X \text{ is either 0 or close to 0 and } p_{i,j} \text{is non null}$$

This representation is base on what the agent experienced with that only trace. There might be a transition between 2 and 1, but it never experienced it for the moment.

- The reward function seems to be deterministic. When the agent leaves s_0, it always receives a reward $r = 1$, when it leaves s_1 it always receives a reward $r = 0$ and when it leaves s_2 it always receives a reward $r = 0$. Hence the reward:

$$R = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

1.2.2

There are two possible way of computing s_0, depending on the assumption done on the trace:

- If we assume that the trace end in a terminal state that is not shown on the graph, then we can use Monte Carlo to do a policy evaluation on s_0 and get the value. According to the program (see **Appendix 1**), Monte Carlo method gives $\hat{V}(s_0) = 1$. With only one episode, this value is quite experimental and gives us a rough estimate. As we saw in class, Monte Carlo method induce a high variance.
- The best way to compute the value of state s_0 since we don't know the end of the episode is to use Temporal Difference method, because we can evaluate the policy with. We have then to suppose that the problem is Markovian. Under this assumption and with a learning rate of $\alpha = 0.3$, TD learning gives $\hat{V}(s_0) = 0.657$.

To conclude, we have to be cautious with these values, because we only have a try with only few states transition, and we don't even know if this is the entire episode, or not.

2 Question 2: Understanding of Grid Worlds

2.1

According to my CID, I have:

$$\boxed{\gamma = 0.35 \text{ and } p = 0.65}$$

2.2

The details of value computation are given in **Appendix 2**. The structure of the code is heavily inspired on the code from lab3. I however changed the way to compute the transition and reward Matrix. I also implemented the algorithm that gives the best policy and the best value function for this problem (function : $value_iteration$).
I used the algorithm of value iteration to get the best policy and value. This is possible under the assumption that the grid world is Markovian. I obtained the results on **Figure 2 and 3**

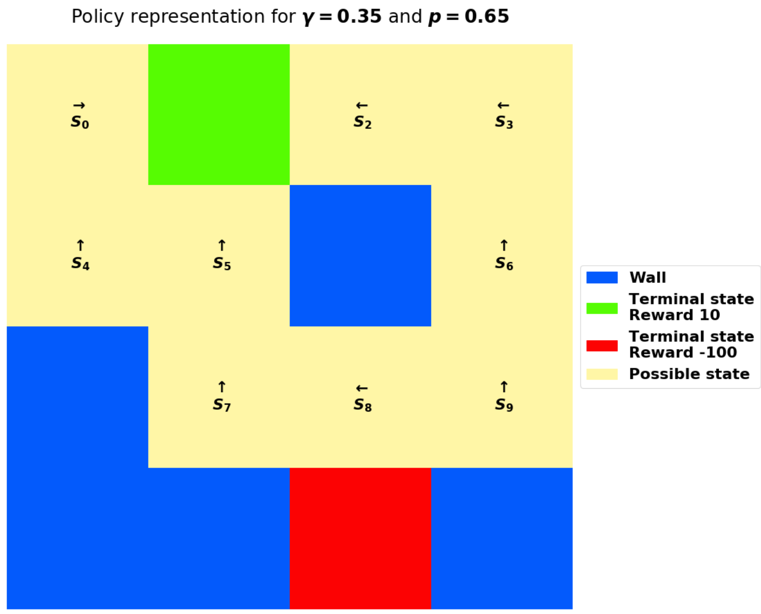

Figure 2: Best policy given

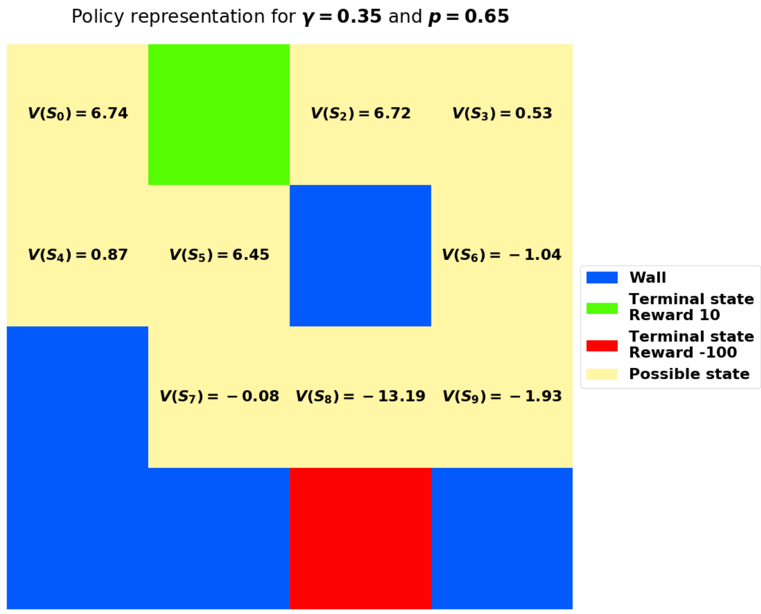

Figure 3: Best Value function

2.3

s_9 is a sensible action, because the agent can reach s_8 that leads to the worst terminal state (with -100 reward). That's why it will try to avoid getting close to the this state and prefer going through other states. If you put $p < 0.25$, then the action will be to go to s_8 since the probability to go to a state that is not s_8 is greater, hecne the arrows get reversed.
The probability $p(a, s_9)$ with the action taken by the optimal policy is 0.65 to go north and 0.11 for the other directions.

2.4

In a world where gamma is equal to zero (ie short-sighted), the agent will actually choose a random path since it cannot remember the previous states it went to. Moreover as discussed before, if $p < 0.25$ then the best policy would actually the opposite policy of the case where $p > 0.25$ because there's a greater chance to get misdirected, hence following the best path **Figure 4**.
The place of the two absorbing would also have change drastically the way the agent behave, and having the two reward states next to each other in s_1 and s_2 can give s_0 the worst reward state.

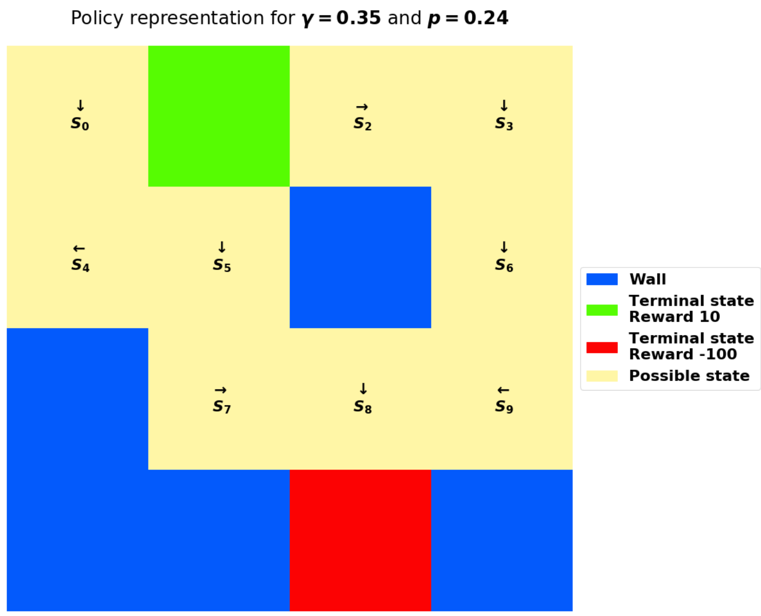

Figure 4: Policy for $p = 0.24$

3 Appendix

```
# Imports
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib.colors import LinearSegmentedColormap
from matplotlib.patches import Patch

# Plot configuration
colors = ["#035afc", "#56fc03", "#fc0303", "#fff6a6"]
values = [1, 2, 3, 4]

norm = mpl.colors.Normalize(vmin=min(values), vmax=max(values))
normed_vals = norm(values)

cmap = LinearSegmentedColormap.from_list(
"mypalette", list(zip(normed_vals, colors)), N=1000
)

vec_func = np.vectorize(lambda l: 4 if l == 0 else l)

font = {"weight": "bold", "size": 22}
mpl.rc("font", **font)
```

3.1 Question 1

3.1.1 Get the trace

```
# Init
CID = "1797836"
gamma = 1
S = ["s1", "s2", "s3"]
cid = list(map(int, [k for k in CID]))

# Get the trace
states = [(k + 2) % 3 for k in cid]
reward = [l % 2 for l in cid]
combined = list(zip(states, reward))
trace = [f"s_{state} {reward} " for state, reward in combined]
print(f"Trace is: {''.join(trace)[:-1]}")
```

3.1.2 Monte Carlo

```
def tomap(l, R):
   R[l].append(reward[l])

R = [[], [], []]
list(map(lambda l: tomap(l, R), states))
v_mc = [np.mean(l) for l in R]

print(
    "".join(
        [
            f"State s_{k} has estimated value {v_mc[k]} with MC\n"
            for k in range(len(v_mc))
        ]
    )
)
```

3.1.3 TD learning

```
v_td = [0, 0, 0]
alpha = 0.3
for k in range(len(states) - 1):
    d = reward[k] + gamma * v_td[states[k + 1]] - v_td[states[k]]
    v_td[states[k]] = v_td[states[k]] + alpha * d


print(
```

```
    "".join(
        [
            f"State s_{k} has estimated value {v_td[k]} with TD\n"
            for k in range(len(v_td))
        ]
    )
)
```

3.2 Question 2

```
# Init
x = 8
y = 3
z = 6
reward_state = ((z + 1) % 3) + 1
reward_loc = (0, 1)

p = 0.25 + 0.5 * x / 10
gamma = 0.2 + 0.5 * y / 10

## Code heavily inspired and adapted from lab3


class GridWorld(object):
    def __init__(self, reward_loc=reward_loc, gamma=gamma, p=p):

        # Size of grid world
        self.shape = (4, 4)

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

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

        # Rewards for each of the absorbing states
        self.special_rewards = {
            reward_loc: 10,
            (3, 2): -100,
        }  # corresponds to each of the absorbing_locs

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

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

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

# Gamma
self.gamma = gamma

# Probability of action
self.p = p
self.p_other = (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 22 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

# 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 rew in self.special_rewards:
    self.rewarders[rew] = 2 if self.special_rewards[rew] > 0 else 3

# 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 value_iteration(self, threshold):
        """
        Value iteration to determine the best policy and the related value
        """
        # 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()

        # Initialise policy
        policy = np.zeros(self.state_size)

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

        # Make a deep copy of the value array to hold the update
        # during the evaluation
        Vnew = np.full(self.state_size, -np.inf)

        # While the Value has not yet converged do:
        while delta > threshold:
            for s in range(self.state_size):

                # Getting the max of value function according to better policy
                tmp_max = -np.inf
                best_action = 0

                for action in range(self.action_size):
                    tmp = 0
                    for s_posterior in range(self.state_size):
                        tmp += T[s_posterior, s, action] * (
                            R[s_posterior, s, action]
```

```
                        + self.gamma * V[s_posterior]
                    )
                if tmp >= tmp_max:
                    best_action = action
                    tmp_max = tmp

            Vnew[s] = tmp_max
            policy[s] = best_action

        #           print("---")
        #           print(f"Vnew = {Vnew}")
        #           print(f"V = {V}")
        #           print(f"policy= {policy}")
        #           print("\n\n\n")

        delta = max(abs(Vnew - V))

        V = np.copy(Vnew)

    return V, policy.astype(int)

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

########### Internal Helper Functions #####################
def paint_maps(self):
    plt.figure(figsize=(15, 15))
    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 state number), 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 = self.build_transition_matrix(S, neighbours)

    # Build the reward matrix
    R = self.build_reward_matrix(S, locations)
```

```
        return S, T, R, absorbing, locations

    def build_reward_matrix(self, S, locations):
        """
        Build the reward matrix of GridWorld
        R[posterior, prior, action]
        Example:
            R[11, 10, 2] is the reward from state s10 to s_11
            action 2
        """

        # Build the reward matrix
        R = self.default_reward * np.ones((S, S, 4))

        for loc in self.special_rewards:
            post_state = self.loc_to_state(loc, locations)
            R[post_state, :, :] = self.special_rewards[loc]

        return R

    def build_transition_matrix(self, S, neighbours):
        """
        Build the transition matrix of GridWorld
        """

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

        for prior_state in range(S):
            for action in range(4): # N E S W
                main_action_state = neighbours[prior_state, action]
                T[main_action_state, prior_state, action] += self.p # / occ

                other_action_state = [
                    l
                    for l in neighbours[prior_state]
                    if l != main_action_state
                ]

                if len(other_action_state) != 3:
                    T[main_action_state, prior_state, action] += (
                        3 - len(other_action_state)
                    ) * self.p_other

                for other_state in other_action_state:
                    T[other_state, prior_state, action] += self.p_other
```

```
        # Absorbing states
        T[:, 1] = np.zeros((11, 4))
        T[:, 10] = np.zeros((11, 4))

        return T

    def get_topology(self):
        """
        locs : list of valid locations
        state_neighbours: matrix number_state X number_actions
                          line ==> state
                          column ==> action (N, E, S, W)
        absorbing : list of absobing states
                    index ==> state
                    value ==> if 1 then absorbing state
        """
        height = self.shape[0]
        width = self.shape[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 not absorbing)
                if self.is_location(loc):
                    locs.append(loc)

                    # Get an array with the neighbours of each state, in
                    # terms of locations
                    local_neighbours = [
                        self.get_neighbour(loc, direction)
                        for direction in ["nr", "ea", "so", "we"]
                    ]
                    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] = int(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.astype(int), 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 grid 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 draw_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(figsize=(20, 15))

        plt.imshow(vec_func(self.walls + self.rewarders), cmap=cmap)
        for state, action in enumerate(policy):
            if self.absorbing[0, state]:
                continue
            arrows = [
                r"$\uparrow$",
                r"$\rightarrow$",
                r"$\downarrow$",
                r"$\leftarrow$",
            ]
            action_arrow = f"{arrows[action]}\n $S_{state}$"
            location = self.locs[state]
            plt.text(
                location[1],
                location[0],
                action_arrow,
                ha="center",
                va="center",
            )
        plt.axis("off")
        legend_elements = [
            Patch(facecolor="#035afc", edgecolor="#035afc", label="Wall"),
            Patch(
                facecolor="#56fc03",
                edgecolor="#56fc03",
                label="Terminal state\nReward 10",
            ),
            Patch(
                facecolor="#fc0303",
                edgecolor="#fc0303",
                label="Terminal state\nReward -100",
            ),
            Patch(
```

```
                facecolor="#fff6a6",
                edgecolor="#fff6a6",
                label="Possible state",
            ),
        ]
        plt.legend(
            handles=legend_elements, loc="center left", bbox_to_anchor=(1, 0.5)
        )
        plt.title(
            f"Policy representation for $\gamma={self.gamma}$ and $p={self.p}$",
            y=1.03,
        )
        plt.savefig("template/policy")
        plt.show()

    def draw_value_function(self, V):

        plt.figure(figsize=(20, 15))

        plt.imshow(vec_func(self.walls + self.rewarders), cmap=cmap)
        for state, action in enumerate(V):
            if self.absorbing[0, state]:
                continue
            action_arrow = f"$V(S_{state})={V[state]:.2f}$"
            location = self.locs[state]
            plt.text(
                location[1],
                location[0],
                action_arrow,
                ha="center",
                va="center",
            )
        plt.axis("off")
        legend_elements = [
            Patch(facecolor="#035afc", edgecolor="#035afc", label="Wall"),
            Patch(
                facecolor="#56fc03",
                edgecolor="#56fc03",
                label="Terminal state\nReward 10",
            ),
            Patch(
                facecolor="#fc0303",
                edgecolor="#fc0303",
                label="Terminal state\nReward -100",
            ),
            Patch(
                facecolor="#fff6a6",
                edgecolor="#fff6a6",
```

```
                label="Possible state",
            ),
        ]
        plt.legend(
            handles=legend_elements, loc="center left", bbox_to_anchor=(1, 0.5)
        )
        plt.title(
            f"Policy representation for $\gamma={self.gamma}$ and $p={self.p}$",
            y=1.03,
        )
        plt.savefig("template/value_function")
        plt.show()


# Initiate the gridworld object and get best policy
g = GridWorld()
V, pol = g.value_iteration(1e-5)

# Gives the drawing
g.draw_policy(pol)
g.draw_value_function(V)
```