Imperial College London

COURSEWORK 1

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Reinforcement Learning

Author:
Arnold Cheung (CID: 01184493)

Date: October 29, 2019

1 Understanding of MDPs

1.1

The personalised trace of CID:01184493 is:

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

1.2

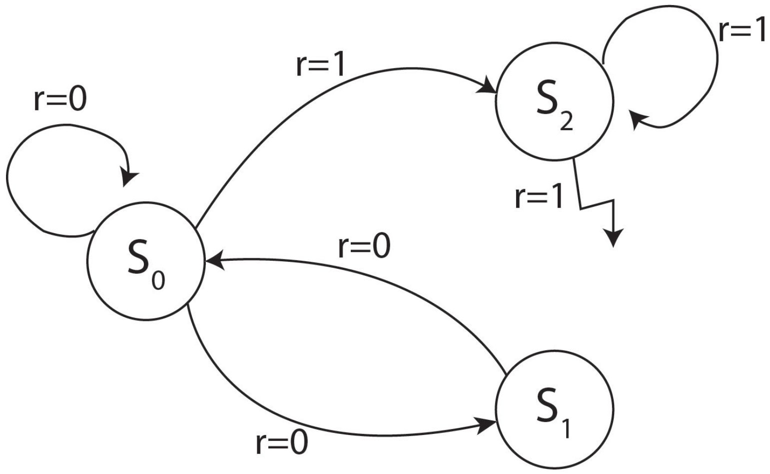

Figure 1: MDP graph of trace τ

Figure one is a minimal MDP graph drawn based on trace τ, three observed states: s_0, s_1 and s_2 are first drawn. The transitions and their respective rewards are then added to the graph, as it is known that they are possible transitions based on the trace. It has been shown in the end of the trace that there is a reward corresponding with a transition out from s_s, it is however not shown which state is the transition leading to, there is therefore a path with reward $r = 1$ out from s_2 but not going into any observed states.

(a) 1. The transition matrix is not deterministic, states can transit to themselves, s0 can go to s0,s1,s2, while s1 has only been shown to be able to go to s0, and s2 has only been shown to go to itself and an unknown state. None of s0, s1, nor s2 are terminal states.

   2. All of the reward shown has been related to s2, either by going into s2, out from s2 or staying in s2.

(b) Most of the reward shown is related to transitions in or out from s_2, it is therefore expected to be the highest value state from what can be observed. s_0 however is also valuable (positive) as it is shown to be able to transit to s_2, despite there is no reward corresponding to transitions to itself or with s_1. Since the trace is not a complete episode, nor is it in a fully known environment with known transition probabilities or rewards, the value of each state can only be estimated using the temporal difference technique (TD). With a single iteration through the trace, assuming the learning rate $\alpha = 0.5$ the value of s_0 is 0 as the trace has not visited s_0 after the firs transition with positive reward. However if the trace is assumed to be traversed twice, the TD algorithm has shown that the value of s_0 has been increased to 1. The code implementation of the algorithm can be found in the appendix.

2 Understanding of Grid Worlds

2.1

The personalised p and γ is:

$$p = 0.45$$
$$\gamma = 0.35$$

2.2

Both the optimal value function and the optimal policy is computed by value iteration with dynamic programming, as the transition probability and reward of every state is known (given a perfect model of the environment as a Markov Decision Process). The algorithm is as follows:

1. Initialise all state values as 0

2. Set the tolerance, θ to a small value (0.01)

3. For each state, s in the environment:

    1. Record the current state value, v
    2. Compute the new state value, $V(s)$:

$$V(s) = max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')] \quad (2)$$

    3. Repeat 3. until the difference magnitude $|v - V(s)|$ is smaller that θ

The optimal state values and policy computed using a Python implementation is shown in Figure 2.

2.3

The optimal policy execute in state s_9 is West (towards s_8) and $p(a, s_9)$ is:

$$p(a, s_9) = \{N : 0.183, W : 0.45, S : 0.183, E : 0.183\} \quad (3)$$

West is a sensible choice of action at s_9 as s_8 is the state with highest value state directly accessible from s_9, since it is closer to the only reward state s_2 than other directly accessible state. As long as γ is greater than 0 (values future reward) and p is greater than 0.25 (probability of successful desired action) both the optimal action and $p(a, s_9)$ will remain the same. Even if the agent values future reward very slightly, the s_8 will always be the state with highest value accessible form s_9, and the probability of executing a desired action will always be the highest out of the four possible actions when $p >= 0.25$.

2.4

One observation from the optimal value and policy is that none of the optimal policy executes are South, this is due to the fact that the penalty state $s_1 1$ is located South relative to every other state, the state optimal policy execute will therefore avoid South to reach a state of higher value. It should also be pointed out that states closer the the reward state s_2 have higher values, with s_1 being the highest at 4.53, slightly higher than the other states that are also directly adjacent to s_2, this is due to the non-deterministic execution of optimal policy, s_1 is surrounded by 2 walls, which means s_1 will arrive back at itself if the actual execution of the policy results in a movement towards the wall, allowing another chance to move into the desired state. The difference in values between adjacent states to s_2 is expected to decrease as p increases (increasingly determinitic execution of policy)

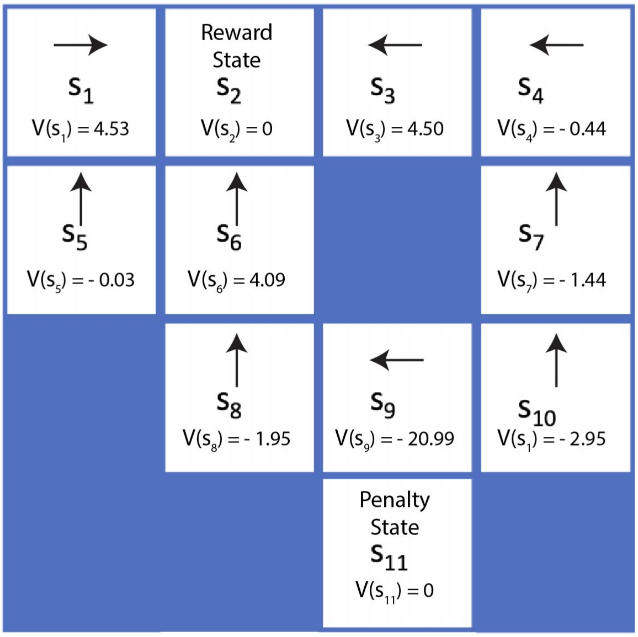

Figure 2: The personalised grid world for CID:01184493. Where s_2 is the reward state and $s_1 1$ is the penalty state. The optimal value of each state is below each state name with the optimal policy at each state above the state name

3 Appendix: Code implementation

3.1 Understanding of MDPs

```
states = {0, 1, 2}  # observed states

trace = [0, 0, 1, 0, 0, 2, 2]  # personalised trace
reward = [0, 0, 0, 0, 0, 1, 1]  # personalised reward


# temporal difference algorithm
def td_estimation(t, r, gamma, alpha):

    V = {0: 0, 1: 0, 2: 2}

    for _ in range(1):
        for step in range(len(t) - 1):
            delta = r[step] + gamma * V[t[step + 1]] - V[t[step]]
            V[t[step]] += alpha * delta

    return V


print(td_estimation(trace, reward, 1, 0.5))
```

3.2 Understanding of Grid Worlds

```
import numpy as np

num_rows = 4
num_cols = 4
reward_states = [(0, 1)]  # define reward state position
penalty_states = [(3, 2)]  # define penalty state poistion

walls = [(1, 2), (2, 0), (3, 0), (3, 1), (3, 3)]  # define walls position

# The environment
class Game:
    def __init__(self, rows, cols, walls, reward_states, penalty_states,
        start_pos, action_success_prob):
        self.rows = rows
        self.cols = cols

        self.board = np.ndarray((rows, cols), dtype=object)
        for wall in walls:
            self.board[wall] = -1  # mark walls (-1)

        for reward_state in reward_states:
            self.board[reward_state] = 1  # mark reward state (1)

        for penalty_state in penalty_states:
            self.board[penalty_state] = 2  # mark penalty state (2)

        state_num = 1

        # replace markings with state state objects
        for i in range(self.rows):
            for j in range(self.cols):
                if not self.board[i, j]:  # normal states are unmarked
                    self.board[i, j] = Position(state_num, -1, is_state=
                        True)
                    state_num += 1
                elif self.board[i, j] == -1:
                    self.board[i, j] = Position(None, None, is_wall=True)
                elif self.board[i, j] == 1:
                    self.board[i, j] = Position(state_num, 10, is_state=
                        True, is_terminal=True)
                    state_num += 1
                elif self.board[i, j] == 2:
                    self.board[i, j] = Position(state_num, -100, is_state
                        =True, is_terminal=True)
                    state_num += 1

        self.actions = ["N", "S", "W", "E"]  # define possible actions
        self.action_success_prob = action_success_prob  # define p (
            personalised p from CID)

        #  give a starting position if needed (not used in value
            iteration)
        if start_pos:
```

```
            self.current_pos = start_pos

        #  score of the current episode (not used)
        self.score = 0

    # returns the next position for a given current position and
        successfully executed action)
    def get_next_position(self, from_pos, action):

        if action == "N":
            next_pos = (from_pos[0] - 1, from_pos[1])
        elif action == "S":
            next_pos = (from_pos[0] + 1, from_pos[1])
        elif action == "W":
            next_pos = (from_pos[0], from_pos[1] - 1)
        else:
            next_pos = (from_pos[0], from_pos[1] + 1)

        # check if next position is legal
        if (next_pos[0] >= 0) and (next_pos[0] <= 3):
            if (next_pos[1] >= 0) and (next_pos[1] <= 3):
                if self.board[next_pos].is_state:
                    return next_pos

        return from_pos  # returns current position if move is not legal
            (i.e move into a wall)

    # returns the actual executed action (non-deterministic policy
        execution)
    def get_next_action(self, intended_action):
        success = np.random.choice([1, 0], p=[self.action_success_prob, 1
            - self.action_success_prob])
        if success:
            return intended_action
        else:
            remaining_actions = self.actions.remove(intended_action)
            return np.random.choice(remaining_actions)

    # returns the transition reward for moving into a certain next state
    def get_transition_reward(self, next_pos):
        return self.board[next_pos].reward

    # returns a transition probability for value iteration
    def get_transition_prob(self, intended_action, actual_action):
        if intended_action == actual_action:
            return self.action_success_prob
        else:
            return (1 - self.action_success_prob) / (len(self.actions) -
                1)

    # show the grid world
    def show_board(self):
        for i in range(self.rows):
            print('-------------------------')
            out = '| '
```

```
            for j in range(self.cols):
                box = str(self.board[i, j])
                box = box.ljust(3, ' ')
                out += box + ' | '
            print(out)
        print('-------------------------')


# State, Wall or Terminal states
class Position:
    def __init__(self, state_num, reward, is_state=False, is_terminal=
        False, is_wall=False):

        self.is_state = is_state
        self.is_terminal = is_terminal
        self.is_wall = is_wall

        if self.is_terminal:
            self.name = 't' + str(state_num)

        elif self.is_state:
            self.name = 's' + str(state_num)

        self.reward = reward

    def __str__(self):
        if self.is_state:
            return self.name
        elif self.is_wall:
            return ' X '


# the agent for value iteration or other algorithms (not implemented)
class Agent:
    def __init__(self, gamma, alpha, environment):
        self.gamma = gamma  # personalised gamma
        self.alpha = alpha  # learning rate (not used)

        self.environment = environment  # load into the created grid
            world
        self.actions = self.environment.actions  # ["N, "S", "W", "E"]

        # initialise state values as 0
        self.state_values = {}
        for i in range(self.environment.rows):
            for j in range(self.environment.cols):
                self.state_values[(i, j)] = 0

        # initialise policy as North
        self.policy = {}
        for i in range(self.environment.rows):
            for j in range(self.environment.cols):
                self.policy[(i, j)] = 'N'

    # value iteration algorithm
```

```python
    def value_iteration(self):
        tolerance = 0.01  # tolerance
        delta = 1000  # initialise difference to be a large value

        while delta > tolerance:
            delta = 0

            # for each state in the grid if it is a state and not a terminal state
            for i in range(self.environment.rows):
                for j in range(self.environment.cols):
                    if self.environment.board[(i, j)].is_state and not self.environment.board[(i, j)].is_terminal:

                        old_max_value = self.state_values[(i, j)]  # v

                        this_state_values = []  # possible value of the current state for different action

                        for ia in self.actions:  # intended action

                            sum_action_values = 0

                            for aa in self.actions:  # actual action (for different possible next states)

                                next_pos = self.environment.get_next_position((i, j), aa)  # next position
                                trans_prob = self.environment.get_transition_prob(ia, aa)  # transition probability
                                trans_reward = self.environment.get_transition_reward(next_pos)  # transition reward

                                sum_action_values += trans_prob * (trans_reward + self.gamma * self.state_values[next_pos])

                            this_state_values.append(sum_action_values)

                        self.state_values[(i, j)] = max(this_state_values) # update state value
                        self.policy[(i, j)] = self.actions[np.argmax(this_state_values)]  # update policy

                        delta = max(delta, abs(old_max_value - self.state_values[(i, j)])) # calculate difference

                        # self.show_values()

    # show state values
    def show_values(self):
```

```
        for i in range(self.environment.rows):
            print('----------------------------------------')
            out = '| '
            for j in range(self.environment.cols):
                if self.environment.board[(i, j)].is_terminal:
                    box = 'T'
                elif self.environment.board[(i, j)].is_wall:
                    box = 'X'
                else:
                    box = "{:.2f}".format(self.state_values[i, j])

                box = box.ljust(7, ' ')
                out += box + ' | '
            print(out)
        print('----------------------------------------')

    # show optimal policy
    def show_policy(self):
        for i in range(self.environment.rows):
            print('------------------------')
            out = '| '
            for j in range(self.environment.cols):
                if self.environment.board[(i, j)].is_terminal:
                    box = 'T'
                elif self.environment.board[(i, j)].is_wall:
                    box = 'X'
                else:
                    box = self.policy[(i, j)]

                box = box.ljust(3, ' ')
                out += box + ' | '
            print(out)
        print('------------------------')


gridworld = Game(num_rows, num_cols, walls, reward_states, penalty_states
    , None, 0.45)  # initiate gridworld
gridworld.show_board()  # show board
agent = Agent(0.35, None, gridworld)  # initiate the agent
agent.value_iteration()  # value iteration
agent.show_values()  # show state values corresponding to board position
agent.show_policy()  # show optimal policy corresponding to board
    position
```