Simple MDP assignment

By Barnabas Katona

Introduction

In the following document I will showcase my solutions to the exercises in the assignment Simple MDP. All the code can be seen within the git folder provided in the Appendix at: Github. Personally I would like to highlight my solution to assignment 2.2 as I went above and beyond by creating an agent and implementing q-learning (until recently I thought it was required for correct compilation of the said assignment). During the entire assignment I strove to minimise the usage of chatGPT and in case I use it it should be for tedious typing and for creating visualisations. Using this principle I have managed to improve my knowledge of python and MDPs in general significantly.

Assignment 1.1

Version A and B have the same set of actions, states, rewards and allowed actions given a state, but the difference is when we calculate the transitional probabilities. Version A includes the reward that state t+1 has given the action t we took at state t, on the other hand version B does not include the reward in it's transitional probabilities, but instead calculates it separately as a deterministic function.

In the practical terms of the assignment version A would, given state t and action t, calculate the state t+1 and the reward brought in state t+1 simultaneously, meanwhile with version B we would have the transitional probability of St+1 | St, At and the reward based of the reward function separately.

Assignment 1.2

Assignment description

Implement a generic stochastic MDP in Python (version 2).

Answer:

At first I asked chatGPT to write some code for me that I later realised wasn't even correct, so I came to the conclusion that my code in Assignment 1.3 perfectly fits the description of this assignment as well AND I didn't use chatGPT at all to reach those results.

The reason why it fits the structure of version 2 as asked is that the probability of going to State t+1 is conditioned on the current state and action we take, so state t and action t.
The same goes for the result, it is calculated by formulate that takes three values:

- Current state
- Current action
- Next state

They are calculated exactly as:

Trans prob: $P(S_{t+1}, |S_t, A_t)$

Reward prob: $R_t = f(S_t, S_{t+1}, A_t)$

Relevant code snipper:

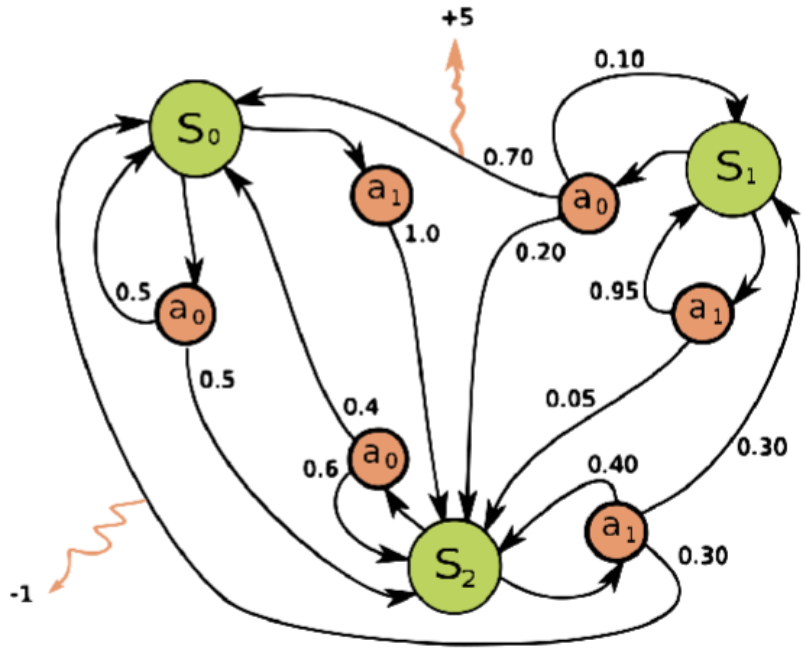
```python
possible_actions = transitions.get(current_state)
if random.uniform(0, 1) <= 0.5:
    chosen_action = 'a'
else:
    chosen_action = 'b'

probabilities = possible_actions[chosen_action]

states = list(probabilities.keys())
weights = list(probabilities.values())

chosen_state = random.choices(states, weights=weights)[0]

if current_state == 's1' and chosen_state == 's0':
    reward = 5
elif current_state == 's2' and chosen_state == 's0':
    reward = -1
else:
   reward = 0
```

For the detailed code please refer to: Code

Assignment 1.3:

Description

Implement the following MDP:



Code

```
import random
import numpy as np

transitions = {
    's0': {
        'a': {'s0': 0.5, 's1': 0.5},
        'b': {'s2': 1.0}
    },
    's1': {
        'a': {'s0': 0.7, 's1': 0.1, 's2': 0.2},
        'b': {'s1': 0.95, 's2': 0.05}
    },
    's2': {
        'a': {'s0': 0.4, 's2': 0.6},
        'b': {'s0': 0.3, 's1': 0.3, 's2': 0.4}
    }
}

rewards = {
    's1': {'a': {'s0': 5}},
    's2': {'b': {'s0': -1}}
}

def next_state(current_state):
    possible_actions = transitions.get(current_state)

    if random.uniform(0, 1) <= 0.5:
        chosen_action = 'a'
    else:
        chosen_action = 'b'

    probabilities = possible_actions[chosen_action]


    states = list(probabilities.keys())
    weights = list(probabilities.values())

    chosen_state = random.choices(states, weights=weights)[0]

    if current_state == 's1' and chosen_state == 's0':
        reward = 5
```

```
    elif current_state == 's2' and chosen_state == 's0':
        reward = -1
    else:
       reward = 0

    print(f"The chosen state is: {chosen_state} with the reward:
{reward}")
    return reward

#Testing s1
current_state = 's1'
rewards = []
for i in range(10):
    rewards.append(next_state(current_state))

print(rewards)
# Testing s2
current_state = 's2'
rewards = []
for i in range(10):

    rewards.append(next_state(current_state))
print(rewards)
```

Explanation of the code

First I inserted the transition probabilities and rewards as given in the assignment description, then I made a function called next_state, which takes the current state as an input.

Each state has two possible actions, a and b, so I made a random float between 0 and 1, if it's less than or equal to 0.5 the action going forward will be ‘a’ otherwise ‘b’.

Then I assign the states list the possible states we can go to from the current state and to weights the chance we have of going to each state. I then choose randomly with the weights given in the ‘weights’ variable.

The function prints the chosen state and the reward received with the step taken.

Output

To test the program I tested first with current state being ‘s1’ then current state being ‘s2’ I ran each 10 times and saved the results in an array each time.

Results for current state = ‘s1’

```
The chosen state is: s1 with the reward: 0
The chosen state is: s0 with the reward: 5
The chosen state is: s1 with the reward: 0
The chosen state is: s1 with the reward: 0
The chosen state is: s1 with the reward: 0
The chosen state is: s1 with the reward: 0
The chosen state is: s0 with the reward: 5
The chosen state is: s0 with the reward: 5
The chosen state is: s1 with the reward: 0
The chosen state is: s1 with the reward: 0
[0, 5, 0, 0, 0, 0, 5, 5, 0, 0]
```

Results for current state = ‘s2’

```
The chosen state is: s1 with the reward: 0
The chosen state is: s0 with the reward: -1
The chosen state is: s0 with the reward: -1
The chosen state is: s2 with the reward: 0
The chosen state is: s1 with the reward: 0
The chosen state is: s0 with the reward: -1
The chosen state is: s2 with the reward: 0
The chosen state is: s2 with the reward: 0
The chosen state is: s1 with the reward: 0
The chosen state is: s2 with the reward: 0
[0, -1, -1, 0, 0, -1, 0, 0, 0, 0]
```

Proof

In order to prove that my program is correct I made a special case where the state is ‘s1’ and the action taken is guaranteed to be ‘a’. Here the chance of going to ‘s0’ is 70%, so I simulated this concept 10.000 times and taken the average of the rewards to see how close it is to 0.7

The code:

```
current_state = 's1'
rewards = []
for i in range(10000):
    rewards.append(next_state(current_state)*2)
print(np.average(rewards))
```

Result

```
Result of proof:
7.015
```

Conclusion, the program does indeed work as intended.

Assignment 2.1

Description

The assignment was to implement the MDP provided in figure 4, whilst making sure the user can choose whether the slipping function is turned on or off and the rate at which slipping should apply.

Assumptions:
For the assignment I assumed the starting position to be (2, 0) as it was between the two ends of the grid and that in case slipping is turned on and it in fact happens the character will move double in the direction it was going to move towards. When that is not possible (as the character would move out of the grid) there will only be one step taken.

figure 4

Code

```
import random

class GridMDP:
    def __init__(self, x, y, slip_rate, slipSwitch):
        self.x = x
        self.y = y
        self.slip = slip_rate
        self.slipSwitch = slipSwitch
        self.start = (2, 0)
        self.end_positive = (x, 0)
        self.end_negative = (0, 0)
```

```
        self.rewards = {(i, 0): 1 if i == x else -1 if i == 0 else 0
for i in range(x + 1)}
        self.actions = ['up', 'down', 'left', 'right']

    def get_reward(self, state):
        return self.rewards.get(state, 0)

    def get_actions(self, state):
        x, y = state
        possible_actions = []
        if x > 0:
            possible_actions.append('left')
        if x < self.x:
            possible_actions.append('right')
        if y > 0:
            possible_actions.append('down')
        if y < self.y:
            possible_actions.append('up')
        return possible_actions

def q_learning(grid, episodes, steps, alpha=0.1, gamma=0.9,
epsilon=0.1):
    q_table = {(i, j): {action: 0 for action in grid.actions} for i in
range(grid.x + 1) for j in range(grid.y + 1)}

    for _ in range(episodes):
        for _ in range(steps):
            state = (random.randint(0, grid.x), random.randint(0,
grid.y))
            if state == grid.end_positive or state ==
grid.end_negative:
                continue

            if random.uniform(0, 1) < epsilon:
                action = random.choice(grid.get_actions(state))
            else:
                action = max(q_table[state], key=q_table[state].get)

            slipping = False
            if grid.slipSwitch:  # Check if slipping is enabled
                if random.uniform(0, 1) < grid.slip:
                    slipping = True
```

```
            # Taking one or two steps based on slipping
            if slipping:
                if action == 'up':
                    next_state = (state[0], min(state[1] + 2, grid.y))
                elif action == 'down':
                    next_state = (state[0], max(state[1] - 2, 0))
                elif action == 'left':
                    next_state = (max(state[0] - 2, 0), state[1])
                elif action == 'right':
                    next_state = (min(state[0] + 2, grid.x), state[1])
            else:
                if action == 'up':
                    next_state = (state[0], min(state[1] + 1, grid.y))
                elif action == 'down':
                    next_state = (state[0], max(state[1] - 1, 0))
                elif action == 'left':
                    next_state = (max(state[0] - 1, 0), state[1])
                elif action == 'right':
                    next_state = (min(state[0] + 1, grid.x), state[1])

            reward = grid.get_reward(next_state)

            q_table[state][action] = q_table[state][action] + alpha * (
                reward + gamma * max(q_table[next_state].values()) -
q_table[state][action]
            )

    return q_table

# Example usage
# Simulate a 1x5 grid with slipping enabled (slip rate 0.2)
grid = GridMDP(4, 0, 0.2, True)
learned_q_table = q_learning(grid, episodes=1000, steps=100)

# Function to print the Q-table
def print_q_table(q_table):
    for state, actions in q_table.items():
        print(f"State: {state}")
        for action, value in actions.items():
            print(f"  Action: {action}, Q-value: {value}")
        print("")

# Print the learned Q-table
```

```
print_q_table(learned_q_table)
```

Explanation of code

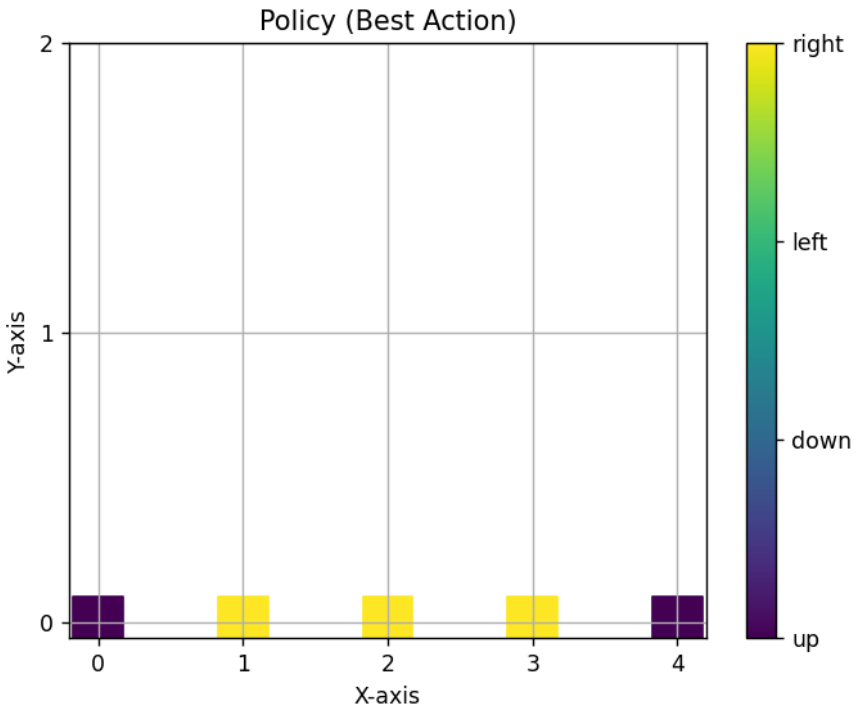
I give a longer explanation in assignment 2.2 since this is a simpler version of that solution (and I did that one first), but the main differences are the grid being fixed 5 x 1 and the starting point, rewards are different, also the up and down actions are blocked across all states.

Output

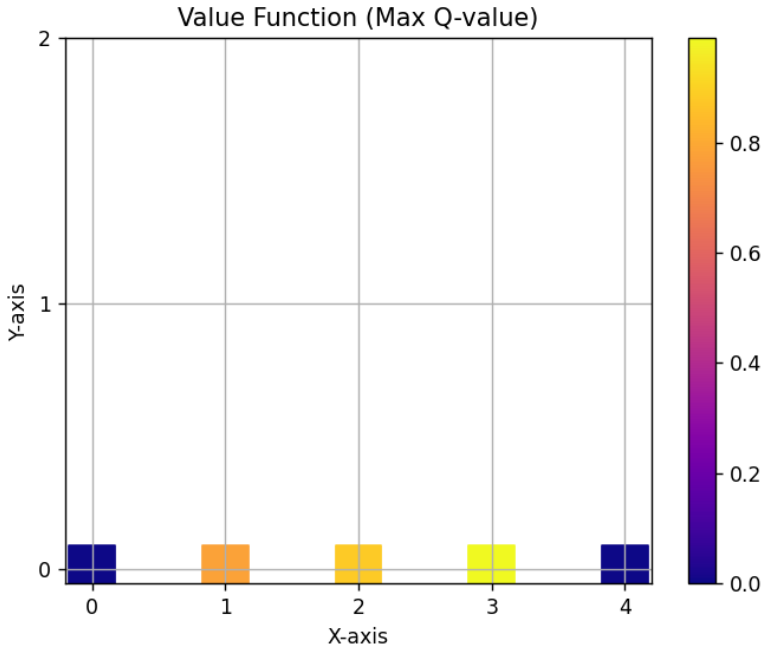
(slipping on, slipping rate = 0.2)

```
State: (0, 0)
  Action: up, Q-value: 0
  Action: down, Q-value: 0
  Action: left, Q-value: 0
  Action: right, Q-value: 0

State: (1, 0)
  Action: up, Q-value: -0.0029701
  Action: down, Q-value: -0.0029701
  Action: left, Q-value: -0.5752281179000001
  Action: right, Q-value: 0.781279916960332

State: (2, 0)
  Action: up, Q-value: -0.00199
  Action: down, Q-value: -0.00199
  Action: left, Q-value: 0.6866356797227013
  Action: right, Q-value: 0.8809999999999991

State: (3, 0)
  Action: up, Q-value: -0.001
  Action: down, Q-value: -0.001
  Action: left, Q-value: 0.7794081665229927
  Action: right, Q-value: 0.9899999999999995

State: (4, 0)
  Action: up, Q-value: 0
  Action: down, Q-value: 0
  Action: left, Q-value: 0
  Action: right, Q-value: 0
```

We start at (2, 0) and as expected going right we have a higher expected reward for each state action pair as we are getting closer to the +1 reward. On the other hand, going left decreases our chances of getting a positive reward, thus decreasing the q values of the state action combinations. In State (1, 0) taking action ‘left’ leads to a guaranteed -1 reward and in state (3, 0) we are guaranteed a +1 reward in case of taking action ‘right’



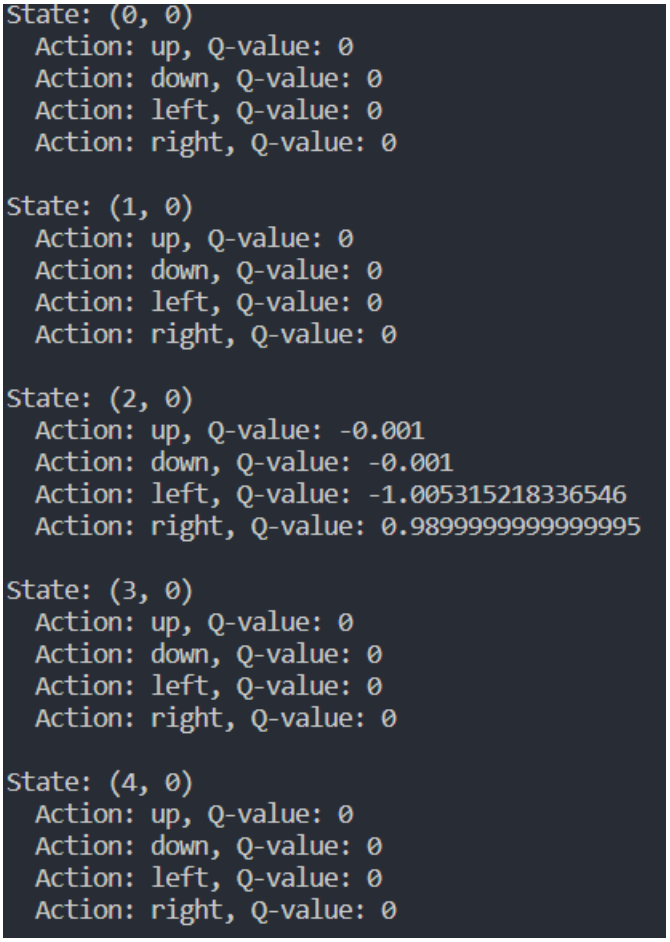
The graph above shows the best actions the agent can take in each state. It does make logical sense, since you always want to go right, except for when you reach an end state (meaning you can't move, thus it shows up as default)



Graph above shows the highest q value for each state. The closer a state is to (4,0) the higher its value is. The reason behind it is that the chance of slipping/ exploring into the dead zone that is (0, 0) is lower

Proof:

The image below proves that the slipping function is working correctly. Since I turned slipping on and put the slipping rate to 100% going either right or left will lead to a guaranteed game ending, this is shown as when we are in our starting state (2, 0) the expected value of taking action ‘left’ is approx -1 and taking action ‘right’ is approx +1. This means that either action will in fact make our character slip and take a double step and the rest of the state have no stay with the base q value, 0, because we can never step on them.



Assignment 2.2

Description

“The current grid size is 3x5. Build the MDP such that it works with an arbitrary integer grid size as input. The bottom

row is always two terminal states one with a positive reward (green) and the rest with negative rewards.”

There is also a picture of the environment provided within the assignment description. See: [Assignment 2.2 picture]

Approach

I first created a class and its constructors then the necessary functions to retrieve the possible actions and rewards associated with given state s. Later I made the Q-learning feature that creates and updates the q table. I used the epsilon greedy algorithm to choose actions for the agent. In the end I made the visualisations.

Assumption: I assumed that the example grid is 5x3 rather than 3x5, because of the picture assigned in the assignment description and since they result the same (the code works for any given x * y grid, I just specify for the following test cases). Same assumptions about the slipperiness as the last assignment.

Code and explanation of code

Construction of class

```
class GridMDP:
    def __init__(self, x, y, slipping_rate, slipSwitch):
        self.x = x
        self.y = y
        self.griddy = (x, y)
        self.start = (0, 0)
        self.slip = slipping_rate
        self.slipSwitch = slipSwitch
        self.end_positive = (x, 0)
        #self.end_negative = [(i, 0) for i in range(1, x)]
        self.bed = [i for i in range(1, x-1)]
        self.actions = ['up', 'down', 'left', 'right']
        self.costOfLife = -0.01
```

The class takes four inputs

- X and Y: these are used to construct the environment grid x * y
- Slipping_rate: this is the chance that the agent is going to slip and take two steps instead of one
- slipSwitch: this boolean determines whether slipperiness is on or off

There are two cases the agent is terminated, when it steps into a negative zone or finishes and gets the positive reward. These states are the variables end_positive and I used self.bed to later identify the bad ending zones. Bed is a list of variables between 0 and x (not including the endings.

The costOfLife is a variable used to incentivize efficient routes for the agent. To put it in perspective I have two graphs in the appendix (costOfLife)with further proof.

Basic functions

```
def get_reward(self, state):
    if state[0] == self.x and state[1] == 0:
        return 1
    elif state[0] in self.bed and state[1] == 0:
        return -1
    else:
        return 0
```

Get_reward function takes a state as an input and returns the reward corresponding to said state.

```
def get_actions(self, state):
    x, y = state
    possible_actions = []
    if x > 0:
        possible_actions.append('left')
    if x < self.x:
        possible_actions.append('right')
    if y > 0:
        possible_actions.append('down')
    if y < self.y:
        possible_actions.append('up')
    return possible_actions
```

.

Get_actions is a function that will make a list of possible actions, this will be used when we want to move the agent.

Q_learning

Q_learning function as a whole (I am going to further break it down and explain it by the chunks):

```python
def q_learning(grid, episodes, steps, alpha=0.1, gamma=0.9, epsilon= 0.2):
    q_table = {(i, j): {action: 0 for action in grid.actions} for i in range(grid.x + 1) for j in range(grid.y + 1)}
    rewPerEpis = []
    #Q = np.zeros([grid.griddy, len(grid.actions)])

    for a in range(episodes):
        state = (0, 0)  # Start at (0, 0)
        totalRev = 0
        # In case the epsilon is high we can decrease it each episode, this way there will be a lot of exploring
        # in the beginning and next to non as the episode numbers increase
        epsilon = epsilon * 0.99
        for b in range(steps):
            reward = 0
            if random.uniform(0, 1) < epsilon:
                action = random.choice(grid.get_actions(state))
            else:

                action = max(q_table[state], key=q_table[state].get)


            # Slipping is turned on when the user signaled it to be so and the random value generated
            # is lower than the chance provided
            slipping = False

            if grid.slipSwitch:
                if random.uniform(0, 1) < grid.slip:
                    slipping = True
```

```python
            # Taking one or two steps based on slipping and possibility.
            # When slipping would make the agent go out of bound it only takes one step instead of two.
            if slipping:
                if action == 'up':
                    next_state = (state[0], min(state[1] + 2 if state[1] + 2 <= grid.y else state[1] + 1, grid.y))
                elif action == 'down':
                    next_state = (state[0], max(state[1] - 2 if state[1] - 2 >= 0 else state[1] - 1, 0))
                elif action == 'left':
                    next_state = (max(state[0] - 2 if state[0] - 2 >= 0 else state[0] - 1, 0), state[1])
                elif action == 'right':
                    next_state = (min(state[0] + 2 if state[0] + 2 <= grid.x else state[0] + 1, grid.x), state[1])
            else:
                if action == 'up':
                    next_state = (state[0], min(state[1] + 1, grid.y))
                elif action == 'down':
                    next_state = (state[0], max(state[1] - 1, 0))
                elif action == 'left':
                    next_state = (max(state[0] - 1, 0), state[1])
                elif action == 'right':
                    next_state = (min(state[0] + 1, grid.x), state[1])
            # Update reward with the new states assigned reward and the costOfLife
            reward = grid.get_reward(next_state) + grid.costOfLife
            # In case the agent goes to a negative termination zone there is an extra punishment, to further encite it to avoid these zones
            if next_state[0] in grid.bed and next_state[1] == 0:
                reward -= 1


            #Q[state, action] += learning_rate * (reward + discount_factor * np.max(Q[new_state, :]) - Q[state, action])

            q_table[state][action] = q_table[state][action] + alpha * (reward + gamma * max(q_table[next_state].values()) - q_table[state][action])

            #Update accumulated rewards for this episode
            totalRev += reward

            # Quit episode in case the new state is a terminal one
            if next_state[0] == grid.x and next_state[1] == 0 or next_state[0] in grid.bed and next_state[1] == 0:
                print(f"Episode {a + 1} finished in steps: {b + 1} with reward {totalRev}")
                break

            # Update the state for the next iteration
            state = next_state

        rewPerEpis.append(totalRev)
    return q_table, rewPerEpis
```

The function called q_learning is where the magic happens, to call it you need to input the following:

- Grid
- Episodes: this is the amount of times the agent should try going through the ‘maze’
- Steps: the amount of steps allowed for each episode

- Alpha: this will be used in the formula used to update the q table. It is important so that an outlier doesn't overwhelm the overall q values
- Gamma: this value is used to decrement the value of future results, as a result more immediate gratifications are preferred. The higher this value the less decreased the future results become and vice-versa. Proof in the appendix at: Gamma
- Epsilon: this value dictates the chance of exploration in the epsilon greedy algorithm. The higher the value the more random the steps taken are.

In the beginning I initialised the q table to be (example is based on 5 x 3 grid and only up until x=4):

```
{
    (0, 0): {'up': 0, 'down': 0, 'left': 0, 'right': 0},
    (0, 1): {'up': 0, 'down': 0, 'left': 0, 'right': 0},
    (0, 2): {'up': 0, 'down': 0, 'left': 0, 'right': 0},
    (1, 0): {'up': 0, 'down': 0, 'left': 0, 'right': 0},
    (1, 1): {'up': 0, 'down': 0, 'left': 0, 'right': 0},
    (1, 2): {'up': 0, 'down': 0, 'left': 0, 'right': 0},
    (2, 0): {'up': 0, 'down': 0, 'left': 0, 'right': 0},
    (2, 1): {'up': 0, 'down': 0, 'left': 0, 'right': 0},
    (2, 2): {'up': 0, 'down': 0, 'left': 0, 'right': 0},
    (3, 0): {'up': 0, 'down': 0, 'left': 0, 'right': 0},
    (3, 1): {'up': 0, 'down': 0, 'left': 0, 'right': 0},
    (3, 2): {'up': 0, 'down': 0, 'left': 0, 'right': 0}
```

I did this by using Nested directory code, where the keys are the states and the actions.

I looped 'episodes' amount of times, each time resetting the state to (0, 0) and the total reward counter. Each episode runs until it's either terminated or it reaches the maximum number of steps. At the beginning I decrease the frequency of the exploration rate by multiplying it with 0.99. I did this so that there is more exploration in the beginning and next to non as the episode number reaches the thousands. This line also allows for more exploration in the beginning by providing an exploration rate that can be close to even 1 and gradually decreasing it more aggressively by decreasing the multiplier of the exploration rate at the beginning of each episode.

Within each episode the q table gets updated and the result of the episode is saved in a variable called rewPerEpis..

Picture of this code snippet in the appendix at: Episode loop code

Each episode:

Epsilon greedy algorithm

```
for b in range(steps):
    reward = 0
    if random.uniform(0, 1) < epsilon:
        action = random.choice(grid.get_actions(state))
    else:

        action = max(q_table[state], key=q_table[state].get)
```

I used the epsilon greedy algorithm to select the action the agent should take. The way this works is by having a number between 0 and 1 being set as the epsilon. This is the variable I called exploration rate until now. Then I used a random number generator in Python to create a number between 0 and 1, in case the number is below said epsilon the agent will take one of the random available steps provided by the get_action(state) function. On the other hand when the previous statement is not true the agent will choose the most optimal (highest q value) action given the state it is in.

Slippery feature

```
slipping = False

if grid.slipSwitch:
    if random.uniform(0, 1) < grid.slip:
        slipping = True
```

I initialise a boolean with value False and update it to True, incase two statements are true:

- User turned the slippery feature on when creating the class object
- The random value generated by Python is lower than the slippery_rate provided by the user

If both requirements are met, slipping is turned on and the agent will take **two** steps in the direction it was otherwise going to take one step towards. I implemented later a safety feature that makes it so the agent can not slip out of the grid.

See appendix for code snippet: Slipping code snippet

Updating the next_state

```
if action == 'up':
    next_state = (state[0], min(state[1] + 1, grid.y))
elif action == 'down':
    next_state = (state[0], max(state[1] - 1, 0))
elif action == 'left':
    next_state = (max(state[0] - 1, 0), state[1])
elif action == 'right':
    next_state = (min(state[0] + 1, grid.x), state[1])
```

The code above shows the logic for updating the next state.

Rewards and updating the Q-table:

```
# Update reward with the new states assigned reward and the costOfLife
reward = grid.get_reward(next_state) + grid.costOfLife
# In case the agent goes to a negative termination zone there is an extra punishment, to
if next_state[0] in grid.bed and next_state[1] == 0:
    reward -= 1

#Q[state, action] += learning_rate * (reward + discount_factor * np.max(Q[new_state, :])

q_table[state][action] = q_table[state][action]
+ alpha * (reward + gamma * max(q_table[next_state].values()) - q_table[state][action])
```

For each step we look at the rewards assigned by the environment using the get_reward function and add the negative reward of taking a step. This is where the logic of taking the shortest path possible takes place. I went into further detail in the appendix at: costOfLife.
In case the agent goes to a **negative** terminal state an additional -1 reward is awarded as to further encite optimal pathing.

In order to update the q table I used the formal definition:

$$Q(s,a) = Q(s,a) + \alpha \cdot (R + \gamma \cdot \max_{a'}(Q(s',a')) - Q(s,a))$$

Broken down:

- Q(s,a): the specific q value given state s and action a
- Alpha: the learning rate: making this higher increases the importance of new explorations
- R: reward received from taking action a
- Gamma: discounting future rewards. The lower this value is the more significant immediate results hold
- Max a (Q(s+1, a+1): this looks for the best possible action that becomes available in the state s+1
- -Q(s,a): this is the last step in adjusting the current q value. This difference brings the new q value closer to the *true* q-value

Breaking the episode

```
# Quit episode in case the new state is a terminal one
if next_state[0] == grid.x and next_state[1] == 0 or next_state[0] in grid.bed and next_state[1] == 0:
    print(f"Episode {a + 1} finished in steps: {b + 1} with reward {totalRev}")
    break
```

In case the agent steps in a terminal state the results of the episode is printed then it's broken, marking the end of the current episode loop.

This function returns a list of the total rewards accumulated in each episode and the constructed q table.

The rest of the code is only about plotting, so I will not go into detail about it

Output

In this section I will be explaining the outputs of the code above.

I will start with the Q values for each state action pair:

```
State: (0, 0)
  Action: up, Q-value: 0.5436340999999978
  Action: down, Q-value: -0.018209306240276914
  Action: left, Q-value: -0.018209306240276914
  Action: right, Q-value: -1.6747886478367018

State: (0, 1)
  Action: up, Q-value: -0.009981433450433296
  Action: down, Q-value: 0.12074640327839675
  Action: left, Q-value: -0.012247897700103202
  Action: right, Q-value: 0.6151489999999982

State: (0, 2)
  Action: up, Q-value: -0.012247897700103202
  Action: down, Q-value: 0.09407879701304354
  Action: left, Q-value: -0.012247897700103202
  Action: right, Q-value: -0.011395636656051615

State: (1, 0)
  Action: up, Q-value: 0
  Action: down, Q-value: 0
  Action: left, Q-value: 0
  Action: right, Q-value: 0

State: (1, 1)
  Action: up, Q-value: 0.032517864865847526
  Action: down, Q-value: -0.38189999999999996
  Action: left, Q-value: 0.16376795582462622
  Action: right, Q-value: 0.6946099999999984

State: (1, 2)
  Action: up, Q-value: -0.00864827525163591
  Action: down, Q-value: 0.23458815139544734
  Action: left, Q-value: -0.008479037702240785
  Action: right, Q-value: -0.008071613638984001
```

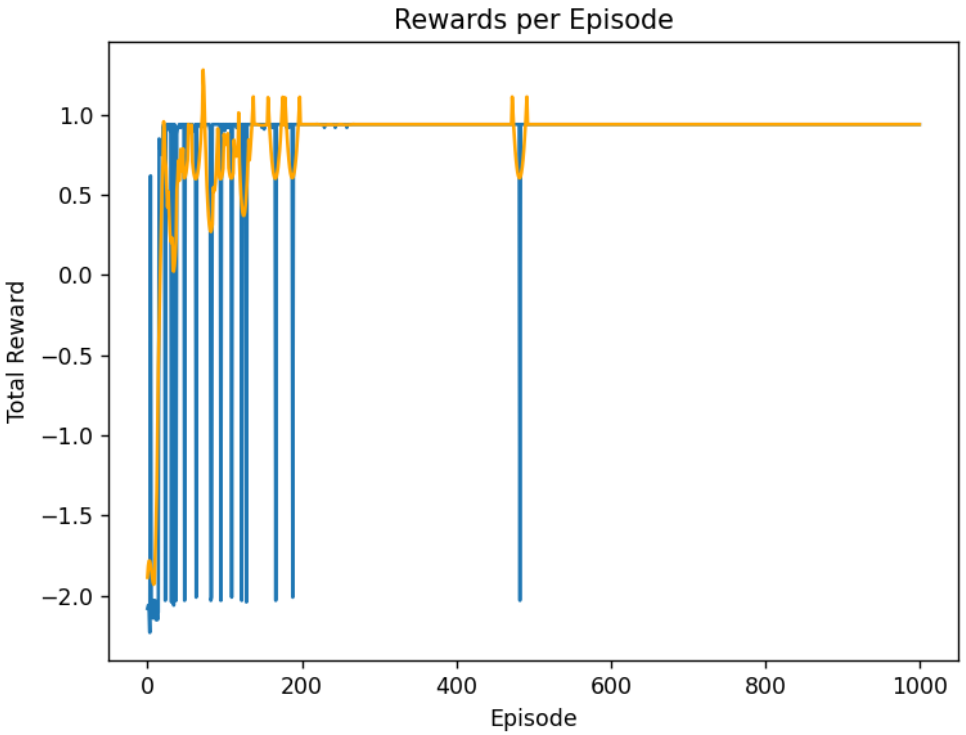
```
State: (2, 0)
  Action: up, Q-value: 0
  Action: down, Q-value: 0
  Action: left, Q-value: 0
  Action: right, Q-value: 0

State: (2, 1)
  Action: up, Q-value: 0.04127785098622764
  Action: down, Q-value: -1.048623231
  Action: left, Q-value: 0.1565556641179346
  Action: right, Q-value: 0.7828999999999988

State: (2, 2)
  Action: up, Q-value: -0.00490099501
  Action: down, Q-value: 0.2753920311351641
  Action: left, Q-value: -0.00552078591564709
  Action: right, Q-value: -0.004509019

State: (3, 0)
  Action: up, Q-value: -0.00109
  Action: down, Q-value: -0.001
  Action: left, Q-value: -0.20099999999999998
  Action: right, Q-value: 0.26829000000000003

State: (3, 1)
  Action: up, Q-value: -0.0034075873771868192
  Action: down, Q-value: 0.020852900000000004
  Action: left, Q-value: 0.06751249136600117
  Action: right, Q-value: 0.8809999999999991

State: (3, 2)
  Action: up, Q-value: -0.0029701
  Action: down, Q-value: -0.002962
  Action: left, Q-value: -0.0032258259099999997
  Action: right, Q-value: 0.06313148154116871
```

```
State: (4, 0)
  Action: up, Q-value: 0
  Action: down, Q-value: 0
  Action: left, Q-value: 0
  Action: right, Q-value: 0

State: (4, 1)
  Action: up, Q-value: 0.13271487125438153
  Action: down, Q-value: 0.9899999999999995
  Action: left, Q-value: 0.14550681527803835
  Action: right, Q-value: 0

State: (4, 2)
  Action: up, Q-value: -0.00199
  Action: down, Q-value: 0.5449976395732362
  Action: left, Q-value: -0.0021601
  Action: right, Q-value: -0.00199
```

Observation from the three images above:

The negative and positive rewards behave as intended. When the agent wants to go to a dead end state the expected return of that action is indeed less than -1 and when the agent wants to go to the goal he can do so while receiving a reward close to 1 (but not exactly, because the cost of living is negative), however this is the least pleasing visualisation to the human eye as it is easy to get lost in it.

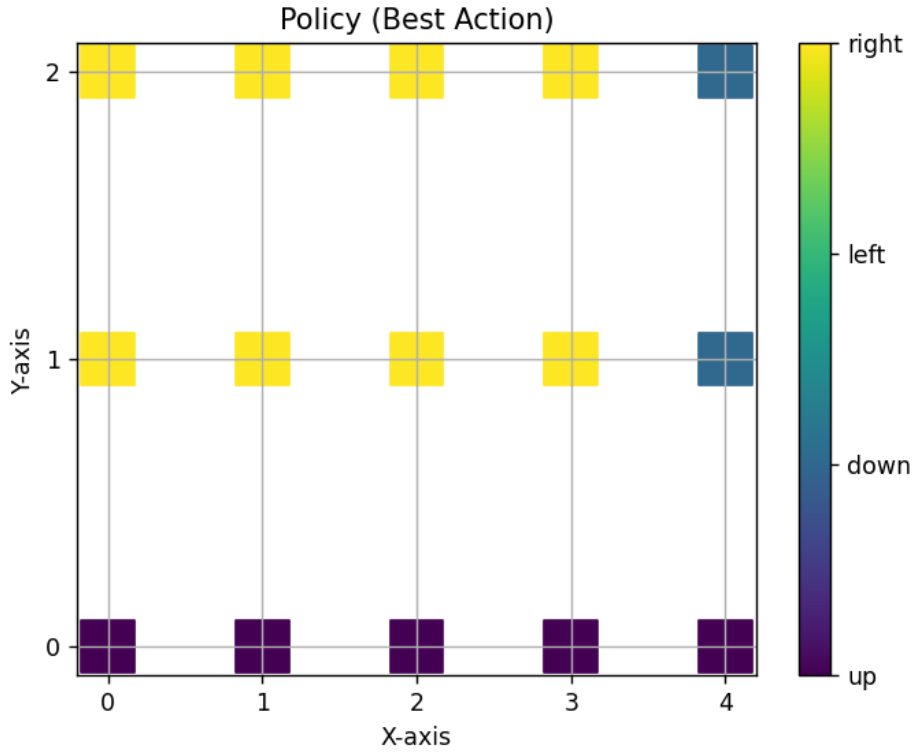
Total rewards per episode:



The image above show the amount of rewards accumulated by each episode before it's termination.
As expected the agent is struggling in the beginning to find the optimal path in the beginning and falls into the negative states quite frequently, but as the q-table gets more accurate and the exploration rate decreases (here epsilon is 0.1 and is being decreased by x0.99 at the beginning of each episode iteration.

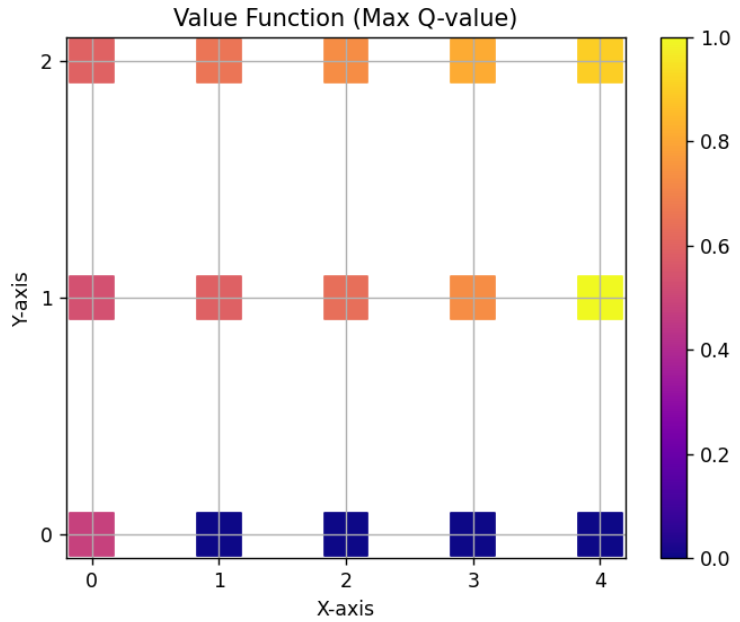
We can see that the agent finds the optimal route at around episode 175.

Best action given state s



The graph above shows the policy of the agent. It does appear to be correct as the best route it can take is {up, right, right, right, right, down}, however as there is no negative reward for taking more steps it doesn't care about taking an alternative, longer route.

Max Q value given state s



This graph above shows the q value of each state. The closer the state is to (4, 0) the higher the expected return as the chance of going into a bad ending is lower and so it the chance of taking extra steps, reducing the total reward.

Assignment 2.3

I will skip it for now, since I have been working on this for too many days straight and I need to pay attention to other submissions too.

Assignment 3.1 and 3.2

3.1/

a)

$V(0,0) = -1$

$V(1,0) = 0 + 0.5 \cdot (V(0,0) \cdot 0,5 + 0.5 \cdot V(2,0))$

$V(2,0) = 0.5(0.5 \cdot V(1,0) + 0.5 \cdot V(3,0)$

$V(3,0) = 0.5(0.5 \cdot V(2,0) + 0.5 \cdot V(4,0))$

$V(4,0) = 1$

b)

$V(0,0) = -1$

$V(1,0) = 0.5 \cdot (V(2,0))$

$V(2,0) = 0.5 \ (V(3,0))$

$V(3,0) = 0.5 \ (V(4,0))$

$V(4,0) = 1$

3.2/

a/ Cost of living: 0.1

$V(0,0) = -1$

$V(1,0) = -0.1 + 0.5(0.5 \cdot V(0,0) + 0.5 \cdot V(2,0))$

$V(2,0) = -0.1 + 0.5(0.5 \cdot V(1,0) + 0.5 \cdot V(3,0))$

$V(3,0) = -0.1 + 0.5(0.5 \cdot V(2,0) + 0.5 \cdot V(4,0))$

$V(4,0) = 1$

b) Cost of living: 2

Same answers as in 3.2/a, except we have to change the -0.1's to -2's

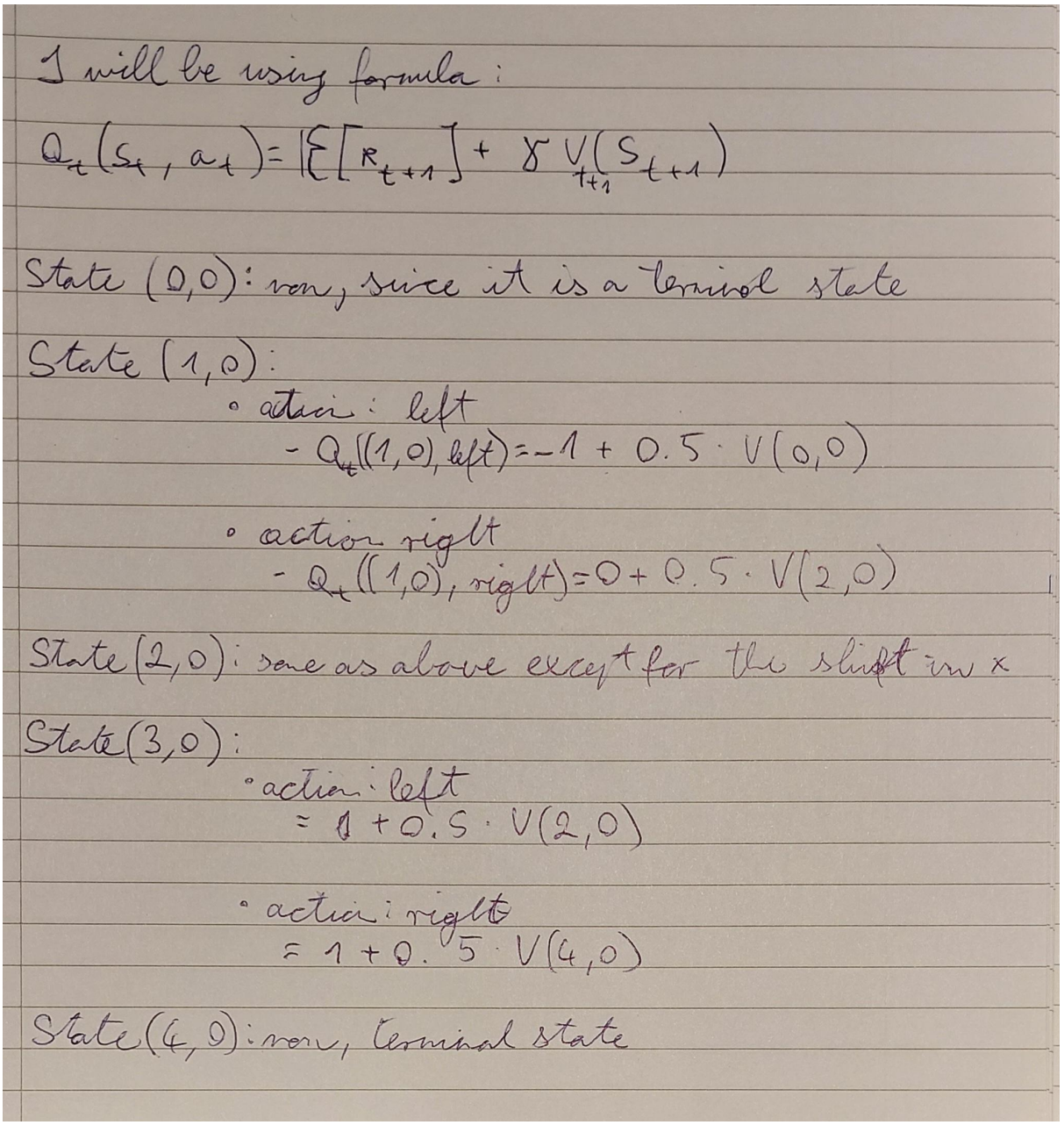
c/ It is worth when the cost of living is -2, because it would take 3 steps to reach the rightmost state, (4,0), which would give

$R = (-2) \cdot 3 + 1 = (-5)$

So going one left is preferable with reward -3

Assignment 3.3

A:



B: *For no cost of living, a deterministic environment, and a decision maker that goes towards the green terminal state.*

Answer: I believe the q values stay the same, however some will stay 0, unless there is exploration involved. For example we wouldn't explore by default Q((2, 0), 'left') as it is going against our policy.

C: *Given your Q values, if the agent changes the policy from what was mentioned to taking the highest Q, how would it change the Q and V function?*

Answer: In the second case, where the policy was already to go straight to the positive rewards there would be no other change than some values potentially being underexplored, as for the first case, where the policy until now was to take actions randomly it would completely transform into the second case.

D: *Reflect on your results, do they make sense?*

Answer: Yes, the two cases becoming the same when the policy is identical is quite logical, since the environment was always the same, the only distinction between the two versions was the policy of the agent

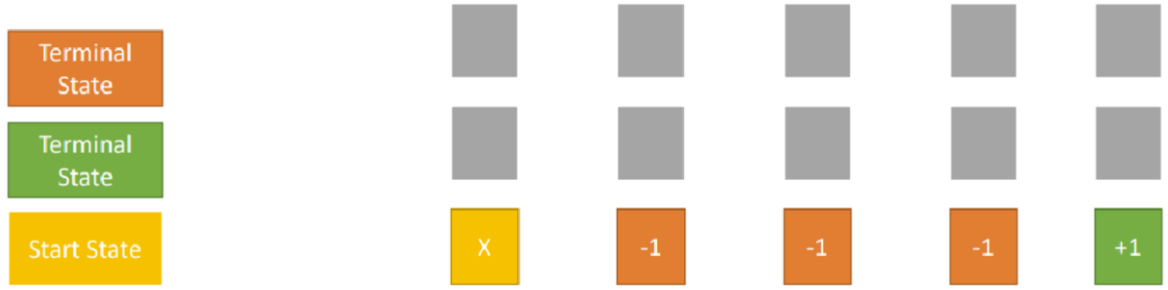
References

The resource I used the most was the slides on the Sharepoint.
LInk (access is limited to people with access to FHICT Fontys):
https://portal.fhict.nl/Studentenplein/LMC/2324nj/Academic_preparations/C-inference/C2.decision-theory/DETH_Slides_v2.pdf

Other references:
https://numpy.org/doc/stable/reference/generated/numpy.zeros.html
https://www.w3schools.com/python/ref_keyword_continue.asp
https://en.wikipedia.org/wiki/Markov_decision_process
https://www.statisticshowto.com/stochastic-model/
https://stackoverflow.com/questions/22805872/optimal-epsilon-%cf%b5-greedy-value
https://gymnasium.farama.org/environments/toy_text/frozen_lake/
https://chat.openai.com/

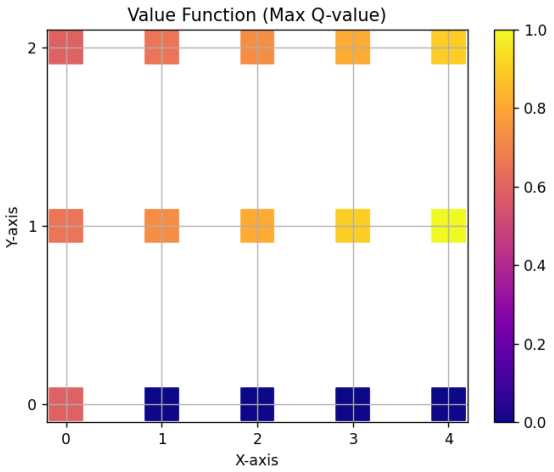
Appendix

Assignment 2.2 picture

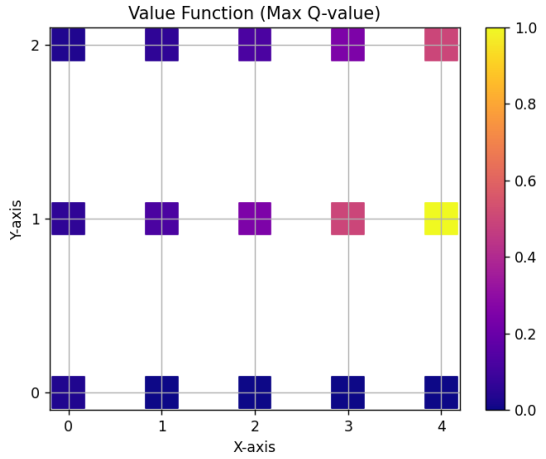


Gamma

Proof that increasing the gamma means more distant rewards are more important:
Gamma = .9



Gamma = .3

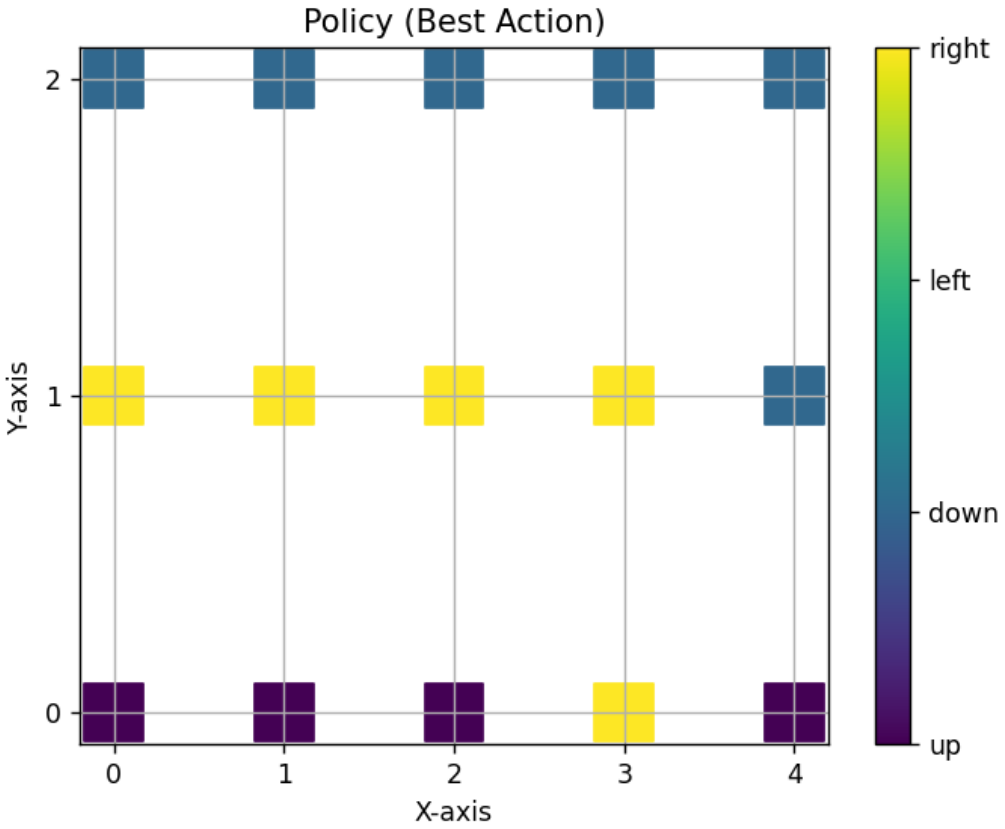


As seen above the further away a state is from the goal (4, 0) the less max q value it has, to the point of being almost 0.

costOfLife

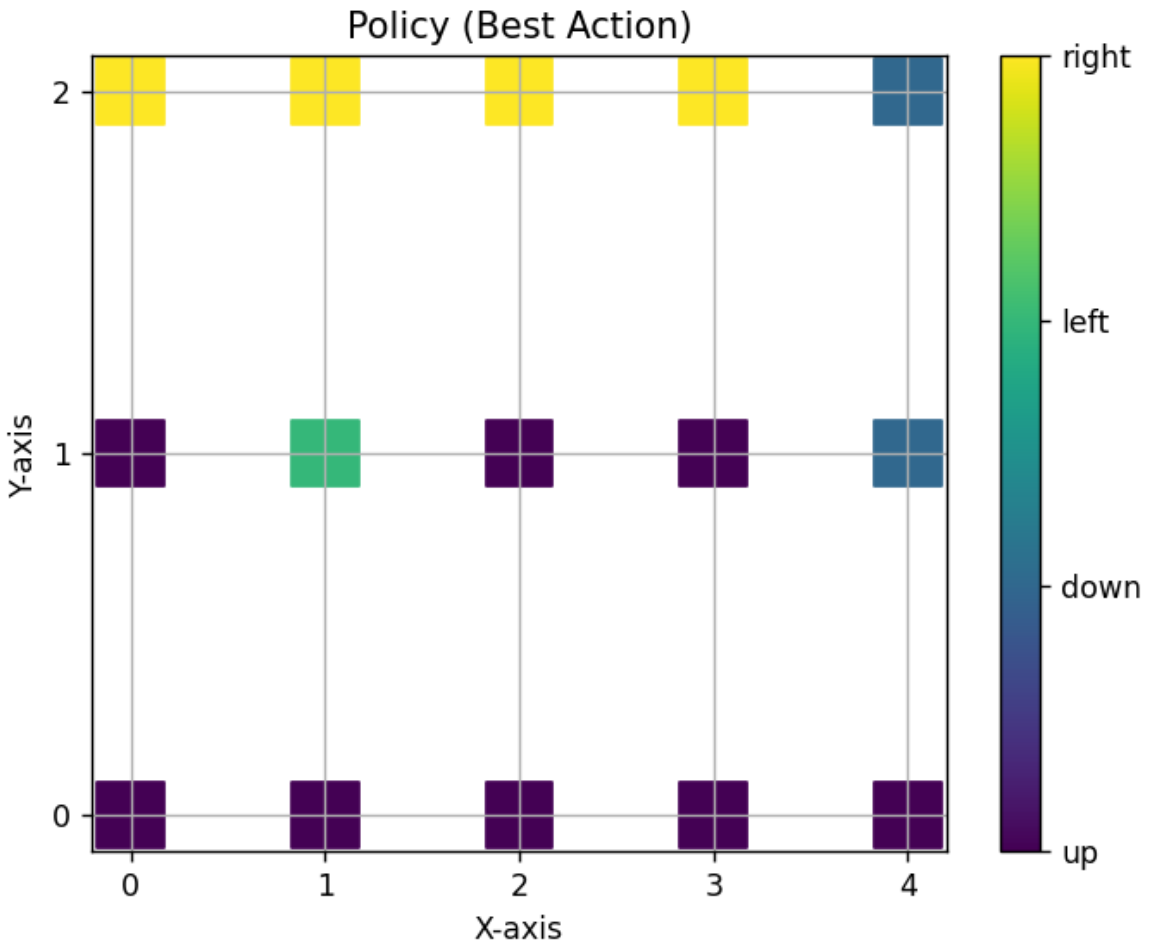
I have two graphs, graph A is the optimal action from each state given costOfLife is turned ON, graph B on the other hand has costOfLife turned off

Graph A:



As seen above the agent takes the shortest route possible from (0, 0) to (4, 0). The path would be: Up - Right - Right - Right - Right - Down which is indeed the case and when there is a deviation due to exploration it will corrugated the path quickly by going down when we go to row 2 or going right when we are in (3, 0) (which is a terminal state, so it doesn't actually make a difference where the agent wants to go since it will reset the environment)

Graph B:



In graph B the path taken is not so obvious. It is shown above that one of the possible paths the agent would take is up- right - left - up -right - right -right -right -down -down, which as you can see is a much larger path. The cause of this is that the agent doesn't get a negative result for taking extra steps.

Nested directory code

```
q_table = {(i, j): {action: 0 for action in grid.actions} for i in range(grid.x + 1) for j in range(grid.y + 1)}
```

Episode loop code

```
for a in range(episodes):
    state = (0, 0)  # Start at (
    totalRev = 0
    # In case the epsilon is high
    # in the beginning and next
    epsilon = epsilon * 0.99
```

Slipping code snippet

```
if slipping:
    if action == 'up':
        next_state = (state[0], min(state[1] + 2 if state[1] + 2 <= grid.y else state[1] + 1, grid.y))
    elif action == 'down':
        next_state = (state[0], max(state[1] - 2 if state[1] - 2 >= 0 else state[1] - 1, 0))
    elif action == 'left':
        next_state = (max(state[0] - 2 if state[0] - 2 >= 0 else state[0] - 1, 0), state[1])
    elif action == 'right':
        next_state = (min(state[0] + 2 if state[0] + 2 <= grid.x else state[0] + 1, grid.x), state[1])
```