CS21D407_CS22E005_PA1_RL

February 24, 2023

1 Authors:

1.0.1 Argha Boksi : CS21D407

1.0.2 Jashaswimalya Acharjee : CS22E005

2 Environment

[1]:
```
from math import floor
import numpy as np
import matplotlib.pyplot as plt

def row_col_to_seq(row_col, num_cols):  #Converts state number to row_column␣
↪format
    return row_col[:,0] * num_cols + row_col[:,1]

def seq_to_col_row(seq, num_cols): #Converts row_column format to state number
    r = floor(seq / num_cols)
    c = seq - r * num_cols
    return np.array([[r, c]])

class GridWorld :
    """
    Creates a gridworld object to pass to an RL algorithm.
    Parameters
    ----------
    num_rows : int
        The number of rows in the gridworld.
    num_cols : int
        The number of cols in the gridworld.
    start_state : numpy array of shape (1, 2), np.array([[row, col]])
        The start state of the gridworld (can only be one start state)
    goal_states : numpy arrany of shape (n, 2)
        The goal states for the gridworld where n is the number of goal
        states.
    """
    def __init__(self, num_rows, num_cols, start_state, goal_states, wind =␣
↪False):
```

```
        self.num_rows = num_rows
        self.num_cols = num_cols
        self.start_state = start_state
        self.goal_states = goal_states
        self.obs_states = None
        self.bad_states = None
        self.num_bad_states = 0
        self.p_good_trans = None
        self.bias = None
        self.r_step = None
        self.r_goal = None
        self.r_dead = None
        self.gamma = 1 # default is no discounting
        self.wind = wind

    def add_obstructions(self, obstructed_states=None, bad_states=None,␣
↪restart_states=None):

        self.obs_states = obstructed_states
        self.bad_states = bad_states
        if bad_states is not None:
            self.num_bad_states = bad_states.shape[0]
        else:
            self.num_bad_states = 0
        self.restart_states = restart_states
        if restart_states is not None:
            self.num_restart_states = restart_states.shape[0]
        else:
            self.num_restart_states = 0

    def add_transition_probability(self, p_good_transition, bias):

        self.p_good_trans = p_good_transition
        self.bias = bias

    def add_rewards(self, step_reward, goal_reward, bad_state_reward=None,␣
↪restart_state_reward = None):

        self.r_step = step_reward
        self.r_goal = goal_reward
        self.r_bad = bad_state_reward
        self.r_restart = restart_state_reward


    def create_gridworld(self):

        self.num_actions = 4
```

```python
        self.num_states = self.num_cols * self.num_rows# +1
        self.start_state_seq = row_col_to_seq(self.start_state, self.num_cols)
        self.goal_states_seq = row_col_to_seq(self.goal_states, self.num_cols)

        # rewards structure
        self.R = self.r_step * np.ones((self.num_states, 1))
        #self.R[self.num_states-1] = 0
        self.R[self.goal_states_seq] = self.r_goal

        for i in range(self.num_bad_states):
            if self.r_bad is None:
                raise Exception("Bad state specified but no reward is given")
            bad_state = row_col_to_seq(self.bad_states[i,:].reshape(1,-1), self.num_cols)
            #print("bad states", bad_state)
            self.R[bad_state, :] = self.r_bad
        for i in range(self.num_restart_states):
            if self.r_restart is None:
                raise Exception("Restart state specified but no reward is given")
            restart_state = row_col_to_seq(self.restart_states[i,:].reshape(1,-1), self.num_cols)
            #print("restart_state", restart_state)
            self.R[restart_state, :] = self.r_restart

        # probability model
        if self.p_good_trans == None:
            raise Exception("Must assign probability and bias terms via the add_transition_probability method.")

        self.P = np.zeros((self.num_states,self.num_states,self.num_actions))
        for action in range(self.num_actions):
            for state in range(self.num_states):


                # check if the state is the goal state or an obstructed state - transition to end
                row_col = seq_to_col_row(state, self.num_cols)
                if self.obs_states is not None:
                    end_states = np.vstack((self.obs_states, self.goal_states))
                else:
                    end_states = self.goal_states

                if any(np.sum(np.abs(end_states-row_col), 1) == 0):
                    self.P[state, state, action] = 1

                # else consider stochastic effects of action
```

```python
            else:
                for dir in range(-1,2,1):

                    direction = self._get_direction(action, dir)
                    next_state = self._get_state(state, direction)
                    if dir == 0:
                        prob = self.p_good_trans
                    elif dir == -1:
                        prob = (1 - self.p_good_trans)*(self.bias)
                    elif dir == 1:
                        prob = (1 - self.p_good_trans)*(1-self.bias)

                    self.P[state, next_state, action] += prob

            # make restart states transition back to the start state with
            # probability 1
            if self.restart_states is not None:
                if any(np.sum(np.abs(self.restart_states-row_col),1)==0):
                    next_state = row_col_to_seq(self.start_state, self.
↪num_cols)
                    self.P[state,:,:] = 0
                    self.P[state,next_state,:] = 1
        return self

    def _get_direction(self, action, direction):

        left = [2,3,1,0]
        right = [3,2,0,1]
        if direction == 0:
            new_direction = action
        elif direction == -1:
            new_direction = left[action]
        elif direction == 1:
            new_direction = right[action]
        else:
            raise Exception("getDir received an unspecified case")
        return new_direction

    def _get_state(self, state, direction):

        row_change = [-1,1,0,0]
        col_change = [0,0,-1,1]
        row_col = seq_to_col_row(state, self.num_cols)
        row_col[0,0] += row_change[direction]
        row_col[0,1] += col_change[direction]

        # check for invalid states
```

```
        if self.obs_states is not None:
            if (np.any(row_col < 0) or
                np.any(row_col[:,0] > self.num_rows-1) or
                np.any(row_col[:,1] > self.num_cols-1) or
                np.any(np.sum(abs(self.obs_states - row_col), 1)==0)):
                next_state = state
            else:
                next_state = row_col_to_seq(row_col, self.num_cols)[0]
        else:
            if (np.any(row_col < 0) or
                np.any(row_col[:,0] > self.num_rows-1) or
                np.any(row_col[:,1] > self.num_cols-1)):
                next_state = state
            else:
                next_state = row_col_to_seq(row_col, self.num_cols)[0]

        return next_state

    def reset(self):
      return int(self.start_state_seq)

    def step(self, state, action):
        p, r = 0, np.random.random()
        for next_state in range(self.num_states):

            p += self.P[state, next_state, action]

            if r <= p:
                break

        if(self.wind and np.random.random() < 0.4):

          arr = self.P[next_state, :, 3]
          next_next = np.where(arr == np.amax(arr))
          next_next = next_next[0][0]
          return next_next, self.R[next_next]
        else:
          return next_state, self.R[next_state]
```

3 Function to create the the gridworld environment

[2]:
```
def make_env(start_state, wind, p_good_transition):

  num_cols = 10
  num_rows = 10
```

```python
    obstructions = np.array([[0,7],[1,1],[1,2],[1,3],[1,7],[2,1],[2,3],
                             [2,7],[3,1],[3,3],[3,5],[4,3],[4,5],[4,7],
                             [5,3],[5,7],[5,9],[6,3],[6,9],[7,1],[7,6],
                             [7,7],[7,8],[7,9],[8,1],[8,5],[8,6],[9,1]])

    bad_states = np.array([[1,9],[4,2],[4,4],[7,5],[9,9]])
    restart_states = np.array([[3,7],[8,2]])
    goal_states = np.array([[0,9],[2,2],[8,7]])

    gw = GridWorld(num_rows=num_rows,
                   num_cols=num_cols,
                   start_state=start_state,
                   goal_states=goal_states,
                   wind=wind)

    gw.add_obstructions(obstructed_states=obstructions,
                        bad_states=bad_states,
                        restart_states=restart_states)

    gw.add_rewards(step_reward=-1,
                 goal_reward=10,
                 bad_state_reward=-6,
                 restart_state_reward=-10)

    gw.add_transition_probability(p_good_transition=p_good_transition,
                                  bias=0.5)

    return gw.create_gridworld()
```

4 Epsilon-Greedy action selection strategy

[3]:
```python
def epsilonGreedy(Q_table, state, epsilon):
  number_of_actions = Q_table.shape[1]
  if np.random.rand() < epsilon:
    action = np.random.randint(number_of_actions)
  else:
    action = np.argmax(Q_table[state, :])
  return action
```

5 Softmax action selection strategy

[4]:
```python
def softmax(Q_table, state, beta):
  q = Q_table[state, :] / beta
  probability = np.exp(q - np.max(q))
  probability /= np.sum(probability)
```

```
    number_of_actions = Q_table.shape[1]
    return np.random.choice(number_of_actions, p=probability)
```

6 Implementation : SARSA

[5]:
```
class Sarsa(object):

  def __init__(self, env, action_selection_strategy, action_selection_param,␣
↪lr, gamma, horizon=100):
    self.env = env
    self.action_selection_stategy = action_selection_strategy
    self.action_selection_param = action_selection_param
    self.lr = lr
    self.gamma = gamma
    self.horizon = horizon

  def check_terminal_state(self, state):
    goal_states = self.env.goal_states_seq
    return state in goal_states

  def train(self, number_of_episodes):
    env = self.env
    action_selection_strategy = self.action_selection_stategy
    action_selection_param = self.action_selection_param
    lr = self.lr
    gamma = self.gamma
    horizon = self.horizon

    reward_per_episode = np.zeros(number_of_episodes)
    steps_per_episode = np.zeros(number_of_episodes)
    Q = np.zeros((env.num_states, env.num_actions))
    state_visit_count = np.zeros(env.num_states)

    for episode in range(number_of_episodes):

      total_reward = 0
      steps = 0
      state = env.reset()
      action = action_selection_strategy(Q, state, action_selection_param)
      state_visit_count[state] += 1

      while (not self.check_terminal_state(state)) and (steps < horizon):
        next_state, reward = env.step(state, action)
        next_action = action_selection_strategy(Q, next_state,␣
↪action_selection_param)
```

```
        Q[state, action] += lr * (reward + gamma*Q[next_state, next_action] - Q[state, action])
        state, action = next_state, next_action

        steps += 1
        total_reward += reward
        state_visit_count[state] += 1

      reward_per_episode[episode] = total_reward
      steps_per_episode[episode] = steps
    state_visit_count /= number_of_episodes

    return reward_per_episode, steps_per_episode, Q, state_visit_count

  def average_performance(self, number_of_epochs, number_of_episodes):
    Q_average = np.zeros((self.env.num_states, self.env.num_actions))
    state_visit_count_average = np.zeros(self.env.num_states)
    average_reward, average_steps = np.zeros(number_of_episodes), np.zeros(number_of_episodes)

    for epoch in range(number_of_epochs):
      reward, steps, Q, state_visit_count = self.train(number_of_episodes)
      average_reward += (reward - average_reward) / (epoch + 1)
      average_steps += (steps - average_steps) / (epoch + 1)
      Q_average += (Q - Q_average) / (epoch + 1)
      state_visit_count_average += (state_visit_count - state_visit_count_average) / (epoch+ 1 )

    return average_reward, average_steps, Q_average, state_visit_count_average
```

7 Implementation : Q Learning

[6]:
```
class QLearning(object):
  def __init__(self, env, action_selection_strategy, action_selection_param, lr, gamma, horizon=100):
    self.env = env
    self.action_selection_stategy = action_selection_strategy
    self.action_selection_param = action_selection_param
    self.lr = lr
    self.gamma = gamma
    self.horizon = horizon

  def check_terminal_state(self, state):
    goal_states = self.env.goal_states_seq
    return state in goal_states
```

```
def train(self, number_of_episodes):
  env = self.env
  action_selection_strategy = self.action_selection_stategy
  reward_per_episode, steps_per_episode = np.zeros(number_of_episodes), np.
↪zeros(number_of_episodes)
  Q = np.zeros((env.num_states, env.num_actions))
  state_visit_count = np.zeros(env.num_states)
  for episode in range(number_of_episodes):
    total_reward, steps = 0, 0
    state = env.reset()
    action = action_selection_strategy(Q, state, self.action_selection_param)
    state_visit_count[state] += 1
    while (not self.check_terminal_state(state)) and (steps < self.horizon):
      next_state, reward = env.step(state, action)
      next_action = action_selection_strategy(Q, next_state, self.
↪action_selection_param)
      Q[state, action] += self.lr * (reward + self.gamma*Q[next_state,␣
↪next_action] - Q[state, action])
      state, action = next_state, next_action
      steps += 1
      total_reward += reward
      state_visit_count[state] += 1
    reward_per_episode[episode] = total_reward
    steps_per_episode[episode] = steps
  state_visit_count /= number_of_episodes
  return reward_per_episode, steps_per_episode, Q, state_visit_count

def average_performance(self, number_of_epochs, number_of_episodes):
  Q_average = np.zeros((self.env.num_states, self.env.num_actions))
  state_visit_count_average = np.zeros(self.env.num_states)
  average_reward, average_steps = np.zeros(number_of_episodes), np.
↪zeros(number_of_episodes)

  for epoch in range(number_of_epochs):
    reward, steps, Q, state_visit_count = self.train(number_of_episodes)
    average_reward += (reward - average_reward) / (epoch + 1)
    average_steps += (steps - average_steps) / (epoch + 1)
    Q_average += (Q - Q_average) / (epoch + 1)
    state_visit_count_average += (state_visit_count -␣
↪state_visit_count_average) / (epoch+ 1 )

  return average_reward, average_steps, Q_average, state_visit_count_average
```

8 Plot functions

[7]:
```
def average_reward_plot(average_reward, episodes, title):
    fig = plt.figure()
    plt.plot(episodes,average_reward,color='red')
    plt.xlabel('Number of episodes')
    plt.ylabel('Average Reward')
    plt.title(title)
    plt.savefig(f'{title}-avg-rwd.png')
    plt.show()
```

[8]:
```
def average_steps_plot(average_steps, episodes, title):
    fig = plt.figure()
    plt.plot(episodes,average_steps,color='red')
    plt.xlabel('Number of episodes')
    plt.ylabel('Number of steps')
    plt.title(title)
    plt.savefig(f'{title}-avg-step.png')
    plt.show()
```

[9]:
```
def plot_Q(Q, title, message = "Q plot"):
    UP = 0
    DOWN = 1
    LEFT = 2
    RIGHT = 3
    D = np.zeros((10, 10))
    for i in range(10):
        D[i, 10-i-1] = 1
    Q = Q.reshape(10, 10, 4)
    for i in range(4):
        Q[:,:, i] = np.dot(D, Q[:,:, i])
    plt.figure(figsize=(8,8))
    plt.title(f"{title}-{message}")
    plt.pcolor(Q.max(-1), edgecolors='k', linewidths=2)
    plt.colorbar()
    def x_direct(a):
        if a in [UP, DOWN]:
            return 0
        return 1 if a == RIGHT else -1
    def y_direct(a):
        if a in [RIGHT, LEFT]:
            return 0
        return 1 if a == UP else -1
    policy = Q.argmax(-1)
    policyx = np.vectorize(x_direct)(policy)
    policyy = np.vectorize(y_direct)(policy)
    idx = np.indices(policy.shape)
```

```
    plt.quiver(idx[1].ravel()+0.5, idx[0].ravel()+0.5, policyx.ravel(), policyy.
↪ravel(), pivot="middle", color='red')
    plt.xticks([])
    plt.yticks([])
    plt.savefig(f'{title}-Q.png')
    plt.show()
```

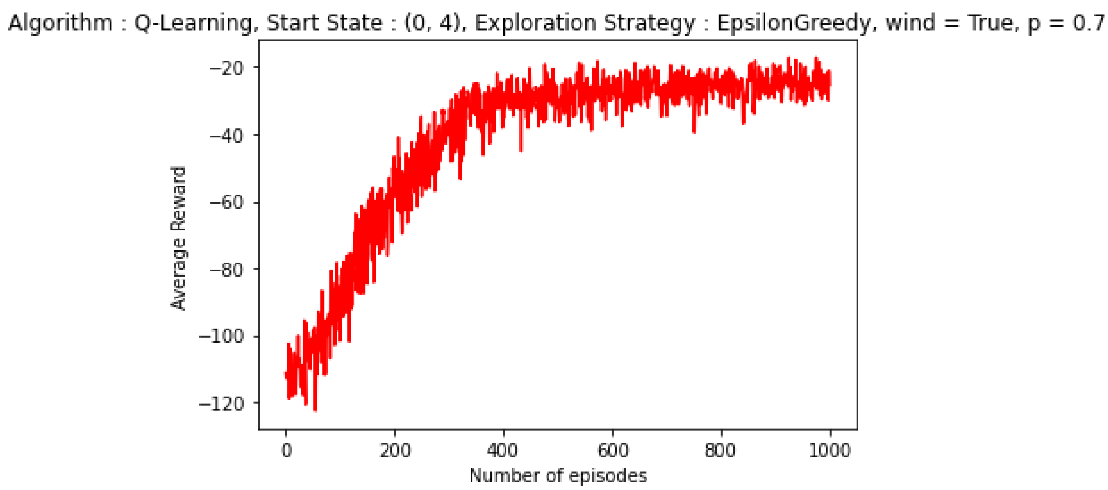
[10]:
```
def plot_step(avg_state_visit_cnt, title):
    D = np.zeros((10, 10))
    for i in range(10):
        D[i, 10-i-1] = 1
    S = np.dot(D, avg_state_visit_cnt.reshape(10, 10))
    plt.figure(figsize=(8,6))
    plt.title(title)
    plt.pcolor(S, cmap="viridis", edgecolors='k', linewidths=2)
    plt.colorbar()
    plt.xticks([])
    plt.yticks([])
    plt.savefig(f'{title}-step.png')
    plt.show()
```

9 Check performance for one particular setting of hyperparateters

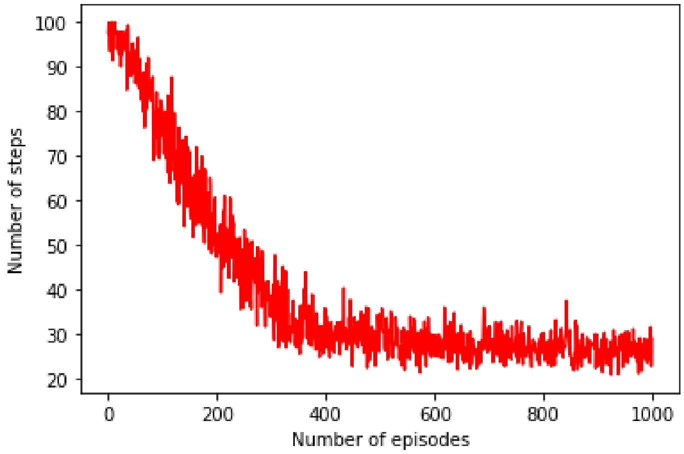
[11]:
```
number_of_epochs = 20
number_of_episodes = 1000
episodes = np.arange(number_of_episodes)
```

[13]:
```
# Name of the learning algorithm to be used : Either Sarsa or Q learning
learning_algo = "Q-Learning"
# Either Epsilon greedy or Softmax
action_selection_strategy = "EpsilonGreedy"
# Specify start state : either (0,4) or (3,6)
start_state = np.array([[0,4]])
# wind is either True or False
wind = True
# p is either 1.0 or 0.7
p = 0.7
# Specify epsilon value for epsilon greedy or beta value for softmax
action_selection_param = 0.01
# Discount Factor
gamma = 1
# Learning rate
lr = 0.1
# creates the gridworld environment with start state, wind and p
env = make_env(start_state, wind, p)
# Action Selection Function : Either epsilon-greedy or Softmax
```

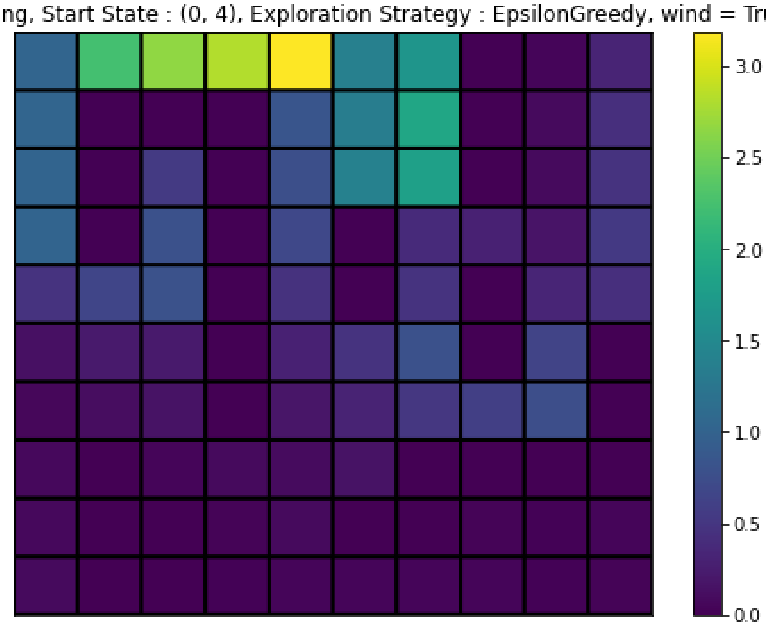
```
action_selection_function = epsilonGreedy
# Select learning algorithm : either Sarsa or Q learning
learning_algorithm = QLearning(env, action_selection_function, action_selection_param, lr, gamma)
# Average performance of the algorithm
average_reward, average_steps, Q_average, state_visit_count_average = learning_algorithm.average_performance(number_of_epochs, number_of_episodes)
# title : a particular hyperparameter configuration
title = "Algorithm : " + learning_algo + ", " + "Start State : " + str(tuple(start_state[0])) + ", " + "Exploration Strategy : " + action_selection_strategy + ", wind = " + str(wind) + ", p = " + str(p)
# average reward vs episodes plot
average_reward_plot(average_reward, episodes, title)
# Number of steps to reach the goal
average_steps_plot(average_steps, episodes, title)
# Heatmap of the grid with state visit counts
plot_step(state_visit_count_average, title)
# Heatmap of the grid with Q values and optimal actions for the best policy
plot_Q(Q_average, title)
```

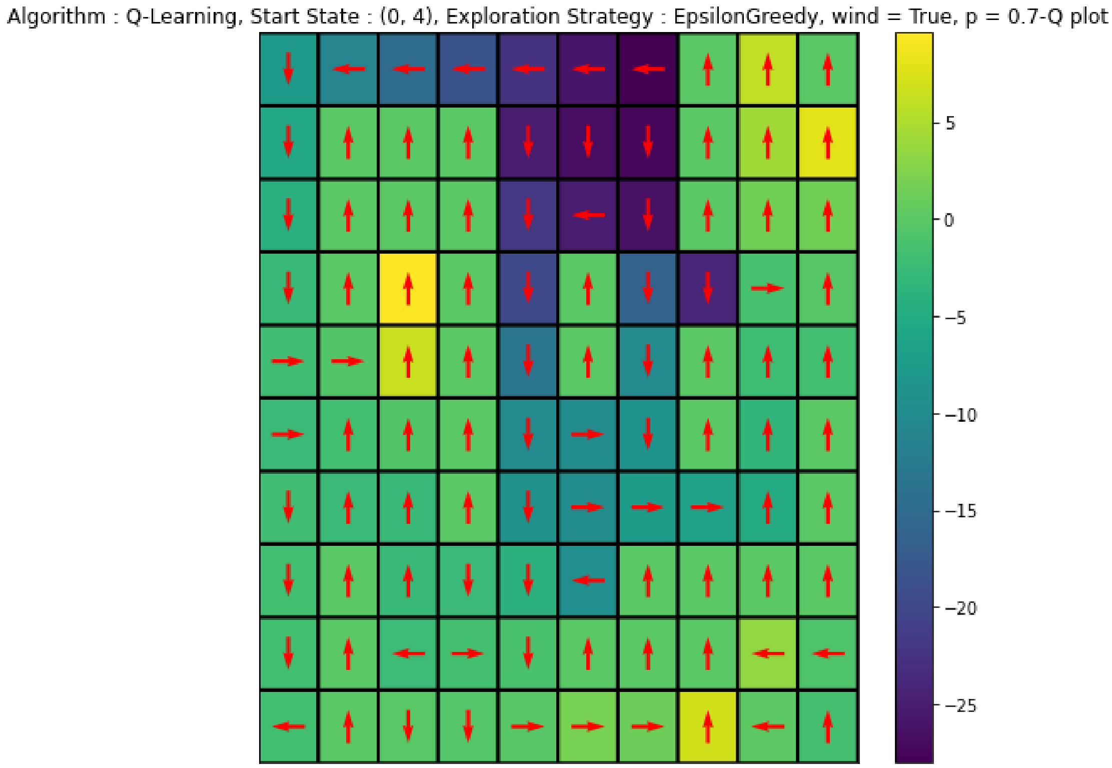


Algorithm : Q-Learning, Start State : (0, 4), Exploration Strategy : EpsilonGreedy, wind = True, p = 0.7



Algorithm : Q-Learning, Start State : (0, 4), Exploration Strategy : EpsilonGreedy, wind = True, p = 0.7





9.0.1 Testing

[12]:
```
# !pip install wandb
# import wandb
# !wandb login
```

[ ]:
```
WANDB_NOTEBOOK_NAME="CS21D407_CS22E005_RL_PA1"
WANDB_RUNS = 20
number_of_epochs = 20
number_of_episodes = 1000
episodes = np.arange(number_of_episodes)

EXPLORE_STRATEGY_MAP = {
    "ep_greedy": epsilonGreedy,
    "softmax": softmax
}

ALGO_MAP = {
    "sarsa": Sarsa,
    "qlearning" : QLearning
}
```

```python
sweep_configuration = {
    'method' : 'bayes',
    'name' : 'sweep',
    'metric' : {
        'goal': 'maximize',
        'name': 'AverageReward'
    },
    'parameters': {
        'lr' : {'values': [0.001, 0.01, 0.1, 1.0]},
        'gamma': {'values': [0.7, 0.8, 0.9, 1.0]},
        'strat_val': {'values': [0.001, 0.01, 0.1]} # EpsGreedy
        # 'strat_val': {'values': [0.01, 0.1, 1, 2]} # Softmax
    }
}

def run_wandb():
  run = wandb.init(project='RL_PA1')

  algos = ["sarsa", "qlearning"]
  explr_strats = ["ep_greedy", "softmax"]
  start_states = ["0_4", "3_6"]
  winds = [False, True]
  p_good_values = [0.7, 1.0]
  lrs = np.array([0.001, 0.01, 0.1, 1.0])
  gammas = np.array([0.7, 0.8, 0.9, 1.0])

  config = dict()
  config["explr"] = "ep_greedy" # Or
  # config["explr"] = "softmax"

  strat_val_epgreedy = np.array([0.001, 0.01, 0.1])
  strat_val_softmax = np.array([0.01, 0.1, 1, 2])

  config = dict()

  for algo in algos:
    config["algo"] = algo
    for start_state in start_states:
      config["start_state"] = start_state
      for wind in winds:
        config["wind"] = wind
        for p in p_good_values:
          config["p_good_value"] = p
          run_sweeps(config)

def run_sweeps(config):
  print(config)
```

```
    # exit()
    start_state = np.array([[int(x.strip()) for x in config["start_state"].
 ↪split("_")]])
    env = make_env(start_state, config["wind"], config["p_good_value"])
    # learning_algorithm = (ALGO_MAP[config["algo"]])(env,␣
 ↪EXPLORE_STRATEGY_MAP[config["explr"]], wandb.config.strat_val, wandb.config.
 ↪lr, wandb.config.gamma)
    learning_algorithm = Sarsa(env, epsilonGreedy, wandb.config.strat_val, wandb.
 ↪config.lr, wandb.config.gamma)
    average_reward, average_steps, Q_average, state_visit_count_average =␣
 ↪learning_algorithm.average_performance(number_of_epochs, number_of_episodes)


    wandb.log({
        'AverageReward': average_reward,
        'AverageSteps': average_steps,
        'Q_average': Q_average,
        'StateVisitCountAverage': state_visit_count_average
    })

    # title = str("Algorithm : " + str(config["algo"]) + ", " + "Start State : "␣
 ↪+ str(start_state) + ", " + "Exploration Strategy : " +␣
 ↪str(config["strat_val"]) + ", wind = " + str(config["wind"]) + ", p = " +␣
 ↪str(config["p_good_value"]))
    # average_reward_plot(average_reward, episodes, title)
    # average_steps_plot(average_steps, episodes, title)
    # plot_step(state_visit_count_average, title)
    # plot_Q(Q_average, title)

sweep_id = wandb.sweep(sweep=sweep_configuration, project='RL_PA1')
wandb.agent(sweep_id, function=run_wandb, count=WANDB_RUNS)

# run_wandb()
```

[15]:

```
from google.colab import drive
drive.mount('/content/drive')

!pip install nbconvert
!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
 ↪texlive-plain-generic
```

[16]:

```
!jupyter nbconvert --to pdf "/content/drive/MyDrive/Colab Notebooks/
 ↪CS21D407_CS22E005_PA1_RL.ipynb"
```

```
[NbConvertApp] Converting notebook /content/drive/MyDrive/Colab
Notebooks/CS21D407_CS22E005_PA1_RL.ipynb to pdf
```

```
[NbConvertApp] Support files will be in CS21D407_CS22E005_PA1_RL_files/
[NbConvertApp] Making directory ./CS21D407_CS22E005_PA1_RL_files
[NbConvertApp] Making directory ./CS21D407_CS22E005_PA1_RL_files
[NbConvertApp] Making directory ./CS21D407_CS22E005_PA1_RL_files
[NbConvertApp] Making directory ./CS21D407_CS22E005_PA1_RL_files
[NbConvertApp] Writing 92053 bytes to ./notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', './notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', './notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 170337 bytes to /content/drive/MyDrive/Colab
Notebooks/CS21D407_CS22E005_PA1_RL.pdf
```

[ ]:

```

```