ME19B79 and ME19B177

Code for Training

In [2]:

```
from math import floor
import numpy as np
from IPython.display import clear_output
import seaborn as sns
sns.set_style('whitegrid')
from typing import Tuple, Optional
```

Grid World Simulation

In [3]:

```
# From the tutorial

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

def seq_to_col_row(seq, num_cols): # Converts state number to row_column forma
    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 = Fa
        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, restar

        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, res

        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
        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)

        self.R = self.r_step * np.ones((self.num_states, 1))
        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), sel
            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 give
            restart_state = row_col_to_seq(self.restart_states[i,:].reshape(1,
            self.R[restart_state, :] = self.r_restart

        if self.p_good_trans == None:
            raise Exception("Must assign probability and bias terms via the ac

        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):
                row_col = seq_to_col_row(state, self.num_cols).reshape(1, -1)
                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:
                    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
                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
                        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) -> int:
      return int(self.start_state_seq)

    def step(self, state, action) -> Tuple[int, float, bool]:
        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_state = next_next[0][0]

        done = state in self.goal_states_seq

        return next_state, self.R[next_state], done


    def rowcol_to_seq(self, row_col: np.ndarray) -> int:  # Converts row_colum
        return row_col[:,0] * self.num_cols + row_col[:,1]

    def seq_to_rowcol(self, seq: int) -> np.ndarray: # Converts state number t
        r = floor(seq / self.num_cols)
        c = seq - r * self.num_cols
        return np.array([r, c])
```

In [4]:

```
from time import time
from sys import stderr
```

In [5]:

```
def sarsa_final(env, Q, gamma, choose_action, alpha, episodes, t_limit=60*3):

    episode_rewards = np.zeros(episodes)
    steps_to_completion = np.zeros(episodes)
    visited_states = np.zeros(shape=Q.shape[0:2])

    t_start = time()

    for ep in range(episodes):
        tot_reward, steps = 0, 0

        state_seq = env.reset()
        state_rowcol = env.seq_to_rowcol(state_seq)
        action = choose_action(Q, state_rowcol)

        done = False
        while not done:
            # Iterate the simulation
            state_next_seq, reward, done = env.step(state_seq, action)
            tot_reward += reward
            steps += 1
            visited_states[state_rowcol[0], state_rowcol[1]] += 1

            # Find the next action
            state_next_rowcol = env.seq_to_rowcol(state_next_seq)
```

```
            action_next = choose_action(Q, state_next_rowcol)

            # Update the Q Value
            Q[state_rowcol[0], state_rowcol[1], action] += alpha * (
                reward +
                gamma * Q[state_next_rowcol[0], state_next_rowcol[1], action_r
                Q[state_rowcol[0], state_rowcol[1], action]
            )

            state_seq, action = state_next_seq, action_next
            state_rowcol = state_next_rowcol

            if steps > 100 or time() > t_start + t_limit:
                False, None, None, None, None

        episode_rewards[ep] = tot_reward
        steps_to_completion[ep] = steps

    return True, Q, episode_rewards, steps_to_completion, visited_states
```

In [6]:

```
def qlearning_final(env, Q, gamma, choose_action, alpha, episodes, t_limit=60*
    episode_rewards = np.zeros(episodes)
    steps_to_completion = np.zeros(episodes)
    visited_states = np.zeros(shape=Q.shape[0:2])
    t_start = time()

    for ep in range(episodes):
        tot_reward, steps = 0, 0

        # Reset environment
        state_seq = env.reset()
        state_rowcol = env.seq_to_rowcol(state_seq)

        action = choose_action(Q, state_rowcol)

        done = False
        while not done:
            visited_states[state_rowcol[0], state_rowcol[1]] += 1
            state_next_seq, reward, done = env.step(state_seq, action)
            state_next_rowcol = env.seq_to_rowcol(state_next_seq)

            tot_reward += reward
            steps += 1

            action_next = choose_action(Q, state_next_rowcol)

            # Update Equation
            Q[state_rowcol[0], state_rowcol[1], action] += alpha * (
                reward
                + gamma * max(Q[state_next_rowcol[0], state_next_rowcol[1]])
                - Q[state_rowcol[0], state_rowcol[1], action]
            )

            state_seq, state_rowcol, action = state_next_seq, state_next_rowcc

            if steps > 100 or time() > t_start + t_limit:
                False, None, None, None, None

        episode_rewards[ep] = tot_reward
```

```
        steps_to_completion[ep] = steps


    return True, Q, episode_rewards, steps_to_completion, visited_states
```

In [7]:

```
from dataclasses import dataclass
from typing import Literal

import numpy as np
from scipy.special import softmax


def epsilon_greedy(self, Q, state):
    epsilon = self.explore_param
    rg = self.rg
    actions = range(len(Q[state[0], state[1]]))
    if rg.rand() < epsilon:
        return rg.choice(actions)
    else:
        return max(actions, key=lambda x: Q[state[0], state[1], x])


def final_choose_action_softmax(self, Q, state):
    tau = self.explore_param
    rg = self.rg
    pis = softmax(Q[state[0], state[1]] / tau)
    assert abs(pis.sum() - 1) < 1e-8
    return rg.choice(range(len(Q[state[0], state[1]])), p=pis)


# All possible test configurations
configs = {
    "method": ("sarsa", "qlearning"),
    "exploration_method": ("EpsilonGreedy", "Softmax",),
    "wind": (False, True),
    "start_state": ((0, 4,), (3, 6,)),
    "alpha": (0.1, 0.2, 0.3, 0.4, 0.5,),
    "gamma": (0.8, 0.9, 0.95, 0.99, 0.999),
    "exploration_value_id": (0, 1, 2, 3, 4),
    "p": (1, 0.7),
}

explore_values_epgreedy = (0, 0.025, 0.05, 0.075, 0.1,)
explore_values_softmax = (0.01, 0.5, 1, 2, 5,)


@dataclass
class Params:
    method: Literal["sarsa", "qlearning"]
    exploration_method: Literal["EpsilonGreedy", "Softmax"]
    wind: bool
    start_state: Literal[(0, 4,), (3, 6,)]

    alpha: float
    gamma: float
    p: float

    exploration_value_id: int
```

```
    def init(self):
        self.explore = self.epsilon_greedy if self.exploration_method == "Epsi
        self.explore_param = explore_values_epgreedy[
            self.exploration_value_id] if self.exploration_method == "EpsilonG
            self.exploration_value_id]
        self.init_env()
        self.method_func = sarsa_final if self.method == "sarsa" else qlearnir
        self.rg = np.random.RandomState(42)

        self.dir_name = f"AA_{self.method}_{self.wind}_{self.start_state[0]}_{
        self.filename = f"AA_{self.alpha}_{self.gamma}_{self.exploration_value

    def epsilon_greedy(self, Q, state):
        epsilon = self.explore_param
        rg = self.rg
        actions = range(len(Q[state[0], state[1]]))
        if rg.rand() < epsilon:  # TODO: eps greedy condition
            return rg.choice(actions)
        else:
            return max(actions, key=lambda x: Q[state[0], state[1], x])

    def softmax(self, Q, state):
        tau = self.explore_param
        rg = self.rg
        pis = softmax(Q[state[0], state[1]] / tau)
        assert abs(pis.sum() - 1) < 1e-8
        return rg.choice(range(len(Q[state[0], state[1]])), p=pis)

    def __str__(self):
        pass


    def init_env(self):
        # specify world parameters
        num_cols = 10
        num_rows = 10

        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]])

        start_state = np.array([self.start_state])
        goal_states = np.array([[0,9],[2,2],[8,7]])

        # create model
        gw = GridWorld(num_rows=num_rows,
                       num_cols=num_cols,
                       start_state=start_state,
                       goal_states=goal_states, wind = self.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=self.p,
                                      bias=0.5)
        self.env = gw.create_gridworld()
```

In [9]:

```
import os
import numpy as np
from pathlib import Path
def write_data(params, r: float, r_s, s_s, visits, q):
    dirs = Path(params.dir_name)
    dirs.mkdir(parents=True, exist_ok=True)
    r = np.array((r,))
    fl = params.dir_name + "/" + params.filename
    if os.path.exists(fl):
        os.remove(fl)
    np.savez(fl, r=r, r_s=r_s, s_s=s_s, visits=visits, q=q)
```

In [10]:

```
def run(parameters: Params):
    # Train the model
    episodes = 1000
    method_func = parameters.method_func
    explore_function = parameters.explore
    gamma = parameters.gamma
    alpha = parameters.alpha
    env = parameters.env

    t_limit = 60 * 3


    def train(t_limit):
        Q = np.zeros((env.num_rows, env.num_cols, env.num_actions))
        success, Q, episode_rewards, steps_to_completion, visited = method_fun
        return success, Q, episode_rewards, steps_to_completion, visited


    # Test the model
    def test(Q, t_limit=10):
        state_seq = env.reset()
        state_rowcol = env.seq_to_rowcol(state_seq)
        action = explore_function(Q, state_rowcol)
        done = False
        steps = 0
        tot_reward = 0
        t_start = time()
        while not done:
            state_seq, reward, done = env.step(state_seq, Q[state_rowcol[0], s
            state_rowcol = env.seq_to_rowcol(state_seq)
            steps += 1
            tot_reward += reward
            if steps > 100 or time() > t_start + t_limit:
                return None
        return tot_reward

    success, Q, episode_rewards, steps_to_completion, visited = train(t_limit)
    if not success:
        stderr.write(f"Train Failed: {parameters.dir_name}/{parameters.filenam
        return None
```

```
    total_reward = test(Q, 10)
    if total_reward is None:
        stderr.write(f"Test Failed: {parameters.dir_name}/{parameters.filename
        return None

    write_data(parameters, total_reward, episode_rewards, steps_to_completion,

    return total_reward
```

In [11]:

```
# Test the params object

c = {
    "method": "qlearning",
    "exploration_method": "EpsilonGreedy",
    "wind": False,
    "start_state": (0, 4,),
    "alpha": 0.1,
    "gamma": 0.9,
    "exploration_value_id": 4,
    "p": 0.7,
}

p = Params(**c)
p.init()
run(p)
```

Out[11]:

```
array([-3.])
```

In [12]:

```
import itertools
from functools import reduce
import operator
from multiprocessing import Pool, cpu_count
import tqdm

def product_dict(**kwargs):
    keys = kwargs.keys()
    vals = kwargs.values()
    for instance in itertools.product(*vals):
        yield dict(zip(keys, instance))

def get_all_params():
    for args in product_dict(**configs):
            pp = Params(**args)
            pp.init()
            yield pp

N = reduce(operator.mul, map(len, configs.values()))

nproc = cpu_count() - 5
print(f"Running with {nproc} cpus")
with Pool(processes=nproc) as p:
    with tqdm.tqdm(total=N) as pbar:
        for _ in p.imap_unordered(run, get_all_params()):
            pbar.update()

print("done")
```

```
Running with 15 cpus
100%|████████████████████████████████████████████████████████████████████████████████| 4000/4000 [2:39:00<00:00,  2.39s/it]Test Failed: AA_sarsa_False_0_4_1_EpsilonGreedy_AA/AA_0.4_0.99_2_AA.npzTest Failed: AA_sarsa_False_0_4_1_EpsilonGreedy_AA/AA_0.4_0.99_4_AA.npzTest Failed: AA_sarsa_False_0_4_1_EpsilonGreedy_AA/AA_0.4_0.999_4_AA.npzTest Failed: AA_sarsa_False_0_4_1_EpsilonGreedy_AA/AA_0.5_0.95_2_AA.npzTest Failed: AA_sarsa_False_0_4_1_EpsilonGreedy_AA/AA_0.5_0.95_3_AA.npzTest Failed: AA_sarsa_False_3_6_1_EpsilonGreedy_AA/AA_0.3_0.999_3_AA.npzTest Failed: AA_sarsa_False_3_6_1_EpsilonGreedy_AA/AA_0.4_0.999_2_AA.npzTest Failed: AA_sarsa_False_3_6_1_EpsilonGreedy_AA/AA_0.5_0.99_2_AA.npzTest Failed: AA_sarsa_False_3_6_1_EpsilonGreedy_AA/AA_0.5_0.999_3_AA.npzTest Failed: AA_sarsa_True_0_4_1_EpsilonGreedy_AA/AA_0.2_0.95_4_AA.npzTest Failed: AA_sarsa_True_0_4_1_EpsilonGreedy_AA/AA_0.3_0.999_4_AA.npzTest Failed: AA_sarsa_True_0_4_1_EpsilonGreedy_AA/AA_0.5_0.99_4_AA.npzTest Failed: AA_sarsa_True_0_4_1_EpsilonGreedy_AA/AA_0.5_0.999_1_AA.npzTest Failed: AA_sarsa_True_0_4_0.7_EpsilonGreedy_AA/AA_0.5_0.999_4_AA.npzTest Failed: AA_sarsa_True_3_6_1_EpsilonGreedy_AA/AA_0.5_0.99_2_AA.npzTest Failed: AA_sarsa_True_3_6_1_EpsilonGreedy_AA/AA_0.5_0.999_3_AA.npzTest Failed: AA_sarsa_False_0_4_0.7_Softmax_AA/AA_0.4_0.8_3_AA.npzTest Failed: AA_sarsa_False_0_4_1_Softmax_AA/AA_0.5_0.8_4_AA.npzTest Failed: AA_sarsa_True_3_6_0.7_EpsilonGreedy_AA/AA_0.5_0.8_1_AA.npzTest Failed: AA_sarsa_True_0_4_1_Softmax_AA/AA_0.1_0.8_4_AA.npzTest Failed: AA_sarsa_True_0_4_1_Softmax_AA/AA_0.1_0.8_2_AA.npzTest Failed: AA_sarsa_True_0_4_0.7_Softmax_AA/AA_0.1_0.8_3_AA.npzTest Failed: AA_sarsa_True_0_4_1_Softmax_AA/AA_0.2_0.8_3_AA.npzTest Failed: AA_sarsa_True_0_4_1_Softmax_AA/AA_0.2_0.8_4_AA.npzTest Failed: AA_sarsa_True_0_4_1_Softmax_AA/AA_0.3_0.8_4_AA.npzTest Failed: AA_sarsa_True_0_4_1_Softmax_AA/AA_0.3_0.8_3_AA.npzTest Failed: AA_sarsa_True_0_4_1_Softmax_AA/AA_0.4_0.8_3_AA.npzTest Failed: AA_sarsa_True_0_4_0.7_Softmax_AA/AA_0.4_0.8_3_AA.npzTest Failed: AA_sarsa_True_0_4_1_Softmax_AA/AA_0.5_0.9_3_AA.npzTest Failed: AA_sarsa_True_0_4_1_Softmax_AA/AA_0.5_0.9_4_AA.npzTest Failed: AA_sarsa_True_0_4_1_Softmax_AA/AA_0.5_0.8_4_AA.npzTest Failed: AA_sarsa_True_0_4_1_Softmax_AA/AA_0.5_0.8_3_AA.npzTest Failed: AA_sarsa_True_3_6_0.7_Softmax_AA/AA_0.1_0.8_1_AA.npzTest Failed: AA_sarsa_True_3_6_0.7_Softmax_AA/AA_0.5_0.8_4_AA.npzTest Failed: AA_qlearning_False_0_4_0.7_EpsilonGreedy_AA/AA_0.4_0.8_1_AA.npzTest Failed: AA_qlearning_True_0_4_1_Softmax_AA/AA_0.1_0.8_3_AA.npzTest Failed: AA_qlearning_True_0_4_1_Softmax_AA/AA_0.1_0.8_4_AA.npzTest Failed: AA_qlearning_True_0_4_1_Softmax_AA/AA_0.2_0.8_3_AA.npzTest Failed: AA_qlearning_True_0_4_1_Softmax_AA/AA_0.2_0.8_4_AA.npzTest Failed: AA_qlearning_True_0_4_1_Softmax_AA/AA_0.3_0.8_4_AA.npzTest Failed: AA_qlearning_True_0_4_0.7_Softmax_AA/AA_0.3_0.8_3_AA.npzTest Failed: AA_qlearning_True_0_4_0.7_Softmax_AA/AA_0.3_0.8_4_AA.npzTest Failed: AA_qlearning_True_0_4_1_Softmax_AA/AA_0.5_0.8_4_AA.npzTest Failed: AA_qlearning_True_0_4_0.7_Softmax_AA/AA_0.5_0.8_4_AA.npz
done
```

In [ ]:

In [14]:
```
!du -sh ./
```

```
108M    ./
```