```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from IPython.display import clear_output
%matplotlib inline
from matplotlib import colors
```

```
from math import floor
import numpy as np

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
        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
                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 render(env, state, ax = None, render_agent = True):
      grid = np.zeros((env.num_rows, env.num_cols))

      for start in start_state:
        grid[start[0], start[1]] = 1 # #0066ff blue color
      for goal in goal_states:
        grid[goal[0], goal[1]] = 2 #66ff66 - green color
      for obs in obstructions:
        grid[obs[0], obs[1]] = 3 #ff3300 - red color
      for bad in bad_states:
        grid[bad[0], bad[1]] = 4 #ffff66 - yellow color
      for restart in restart_states:
        grid[restart[0], restart[1]] = 5 #ff6600 - orange color
      if render_agent:
        grid[seq_to_col_row(state,10)[0], seq_to_col_row(state,10)[1]] = 6 #000000 - black color

      if render_agent:
          cmap = colors.ListedColormap(['#ffffff', '#0066ff', '#66ff66','#ff3300', '#ffff66', '#ff6600', '#000000'])
```

```
      else:
          cmap = colors.ListedColormap(['#ffffff', '#0066ff', '#66ff66','#ff3300', '#ffff66', '#ff6600'])

      if ax is None:
          fig, ax = plt.subplots()
          fig.set_size_inches(10,10)

      ax.pcolor(grid, cmap=cmap, edgecolors='k', linewidths=2)
      return ax

    '''
    render_policy: render a learnt policy
    '''
    def render_policy(self, policy):
        pass

    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]
```

```
# 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([[3,6]])
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 = False)
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=1.0,
                              bias=0.5)
env1 = gw.create_gridworld()
```

```
# 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([[3,6]])
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 = False)
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=0.7,
                              bias=0.5)
env2 = gw.create_gridworld()
```

```
# 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([[0,4]])
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 = False)
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=1.0,
                              bias=0.5)
env3 = gw.create_gridworld()
```

```
# 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([[0,4]])
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 = False)
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=0.7,
                              bias=0.5)
env4 = gw.create_gridworld()
```

```
# 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([[3,6]])
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 = True)
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=1.0,
                              bias=0.5)
env5 = gw.create_gridworld()
```

```
# 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([[3,6]])
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 = True)
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=0.7,
```

```
                              bias=0.5)
env6 = gw.create_gridworld()
```

```
# 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([[0,4]])
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 = True)
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=1.0,
                              bias=0.5)
env7 = gw.create_gridworld()
```

```
# 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([[0,4]])
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 = True)
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=0.7,
                              bias=0.5)
env8 = gw.create_gridworld()
```

```
env = env1 #Environment 1-8
```

```
print("Number of actions", env.num_actions) #0 -> UP, 1-> DOWN, 2 -> LEFT, 3-> RIGHT
print("Number of states", env.num_states)
print("start state", env.start_state_seq)
print("goal state(s)", env.goal_states_seq)
```

```
Number of actions 4
Number of states 100
start state [36]
goal state(s) [ 9 22 87]
```

```
env.P[0,:,0]
```

```
array([1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

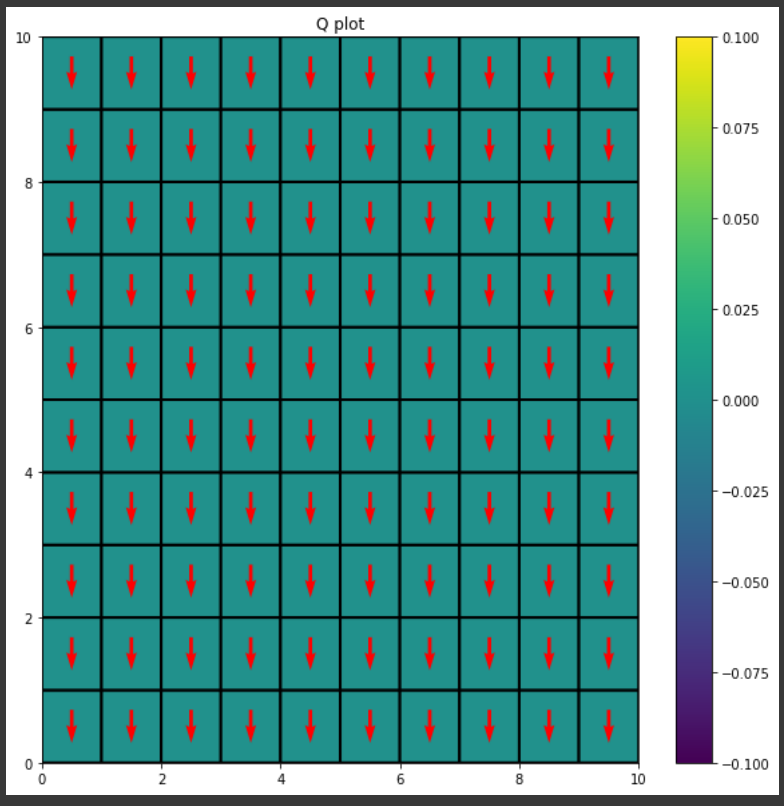
```
DOWN = 0
UP = 1
LEFT = 2
RIGHT = 3
actions = [DOWN, UP, LEFT, RIGHT]
```

Q-value

We can use a 3D array to represent Q values. The first two indices are X, Y coordinates and last index is the action.

```
def plot_Q(Q, message = "Q plot"):
    plt.figure(figsize=(10,10))
    plt.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 = np.zeros((num_rows, num_cols))
    for i in range(num_rows):
        for j in range(num_cols):
            state = i*num_cols + j
            row, col = np.unravel_index(state, (num_rows, num_cols))
            action = np.argmax(Q[row, col])
            policy[i, j] = action
    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.show()
```

```
num_states = num_rows * num_cols
Q = np.zeros((num_rows, num_cols, gw.num_actions))

plot_Q(Q)
```

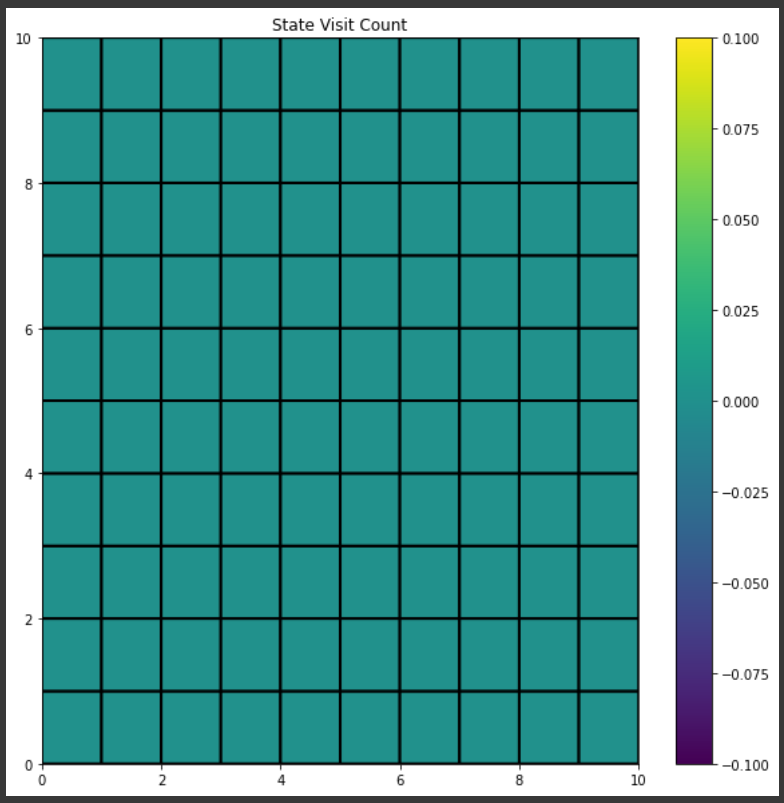


State Visit Counts

Plots the heatmap for the number of times each state was visited throughout the training phase.

```
def plot_state_visits(state_visits, message = "State Visit Count"):
    plt.figure(figsize=(10,10))
    plt.title(message)
    plt.pcolor(state_visits, edgecolors='k', linewidths=2)
    plt.colorbar()
    plt.show()
```

```
state_visits = np.zeros((num_rows, num_cols))

plot_state_visits(state_visits)
```



Exploration strategies

1. Epsilon-greedy
2. Softmax

```
from scipy.special import softmax

seed = 42
rg = np.random.RandomState(seed)

# Epsilon greedy
def choose_action_epsilon(Q, state, epsilon = 0.1, beta = 1, rg=rg):
    #if isinstance(state, int):
        #state = (state // gw.num_cols, state % gw.num_cols)
    if not Q[state].any():
        return rg.choice(Q.shape[-1])
    else:
        if rg.rand() < epsilon:
            return rg.choice(Q.shape[-1])
        else:
            return np.argmax(Q[state])

# Softmax
def choose_action_softmax(Q, state, epsilon = 0.1, beta = 1, rg=rg):
   #if isinstance(state, int):
      # state = (state // gw.num_cols, state % gw.num_cols)
   #probabilities = softmax(Q[state[0], state[1]])
   #print(state)
   #state = 10*state[0] + state[1]
   #print(Q[state].shape)
   probabilities = softmax(Q[state]/beta)
   return rg.choice(np.arange(len(probabilities)), p=probabilities)
```

SARSA

Now we implement the SARSA algorithm.

Recall the update rule for SARSA:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Hyperparameters

So we have some hyperparameters for the algorithm:

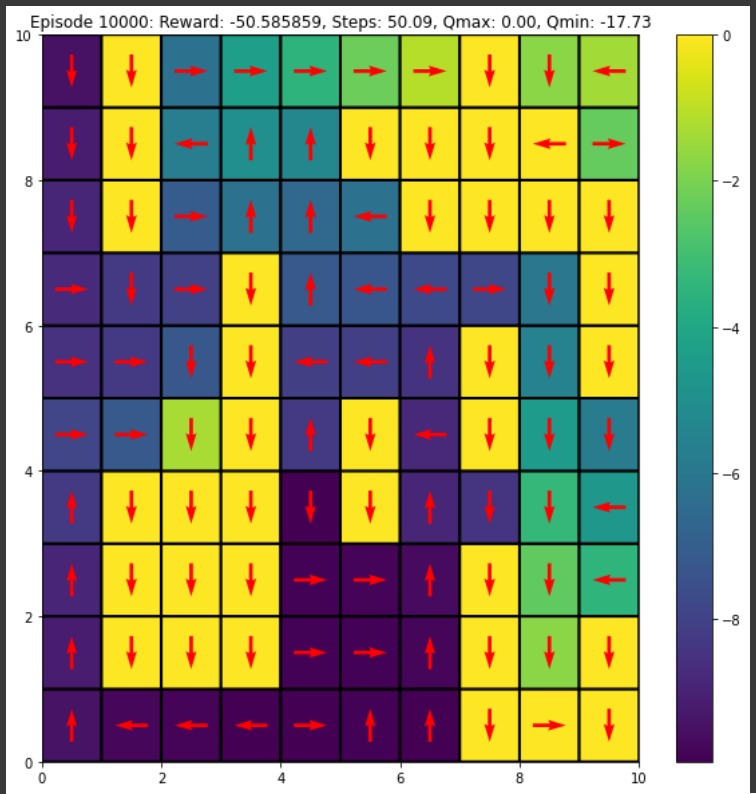
- α (learning rate)
- γ (discount factor)
- ϵ: For epsilon greedy exploration
- β: For Softmax exploration (temperature)

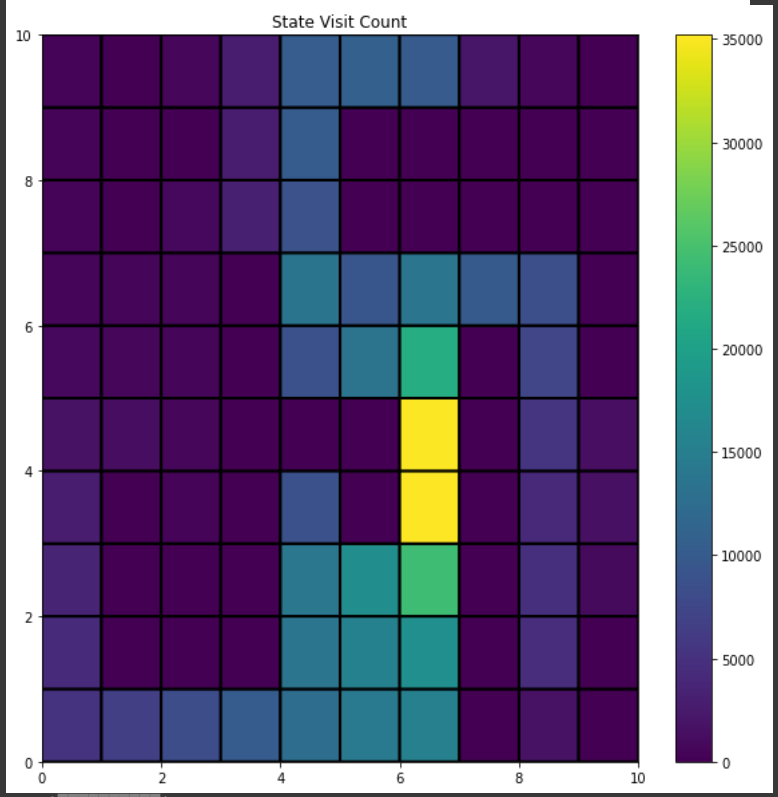
```
# initialize Q-value
Q = np.zeros((gw.num_rows, gw.num_cols, env.num_actions))

alpha0 = 0.4
gamma = 0.9
beta0 = 1
episodes = 10000
#episodes = 500
epsilon0 = 0.1
```

```
print_freq = 100

def sarsa(env, Q, state_visits, alpha=0.4, gamma=0.9, epsilon = 0.1, beta = 1, plot_heat=False, choose_action=choose_action_softmax):
    episode_rewards = np.zeros(episodes)
    steps_to_completion = np.zeros(episodes)

    if plot_heat:
        clear_output(wait=True)
        plot_Q(Q)
        plot_state_visits(state_visits)
    #epsilon = epsilon0
    #alpha = alpha0
    #beta = beta0

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

        # Reset environment
        state = env.reset()
        Q = np.reshape(Q,(100,4))
        state_visits = np.reshape(state_visits,(100))
        #state_int = row_col_to_seq(np.array([state]), env.num_cols)[0]
        action = choose_action(Q, state, epsilon, beta)
        done = False

        while not done:
            state_next, reward = env.step(state, action)

            if (state_next in env.goal_states_seq):
                done = True
                break;

            #state_next_int = row_col_to_seq(np.array([state_next]), env.num_cols)[0]
            action_next = choose_action(Q, state_next,epsilon, beta)

            # Update Q-values
            #print(state)
            #print('qvalue')
            Q[state, action] += alpha * (reward + gamma * Q[state_next, action_next] - Q[state, action])
            #print(state)
            state_visits[state] += 1;
            total_reward += reward
            steps += 1

            state, action = state_next, action_next

        episode_rewards[ep] = total_reward
        steps_to_completion[ep] = steps
        Q = np.reshape(Q,(10,10,4))
        state_visits = np.reshape(state_visits,(10,10))
```

```
        if (ep+1)%print_freq == 0 and plot_heat:
            clear_output(wait=True)
            plot_Q(Q, message = "Episode %d: Reward: %f, Steps: %.2f, Qmax: %.2f, Qmin: %.2f"%(ep+1, np.mean(episode_rewards[ep-print_fre
                                                                                               np.mean(steps_to_completion[ep-print_freq+1:ep]),
                                                                                               np.reshape(Q,(10,10,4)).max(), np.reshape(Q,(10,10,4)).min()))
            plot_state_visits(state_visits)

    return Q, state_visits, episode_rewards, steps_to_completion
```

```
#Q, state_visits, rewards, steps = sarsa(env, Q, state_visits, alpha=0.4, gamma=0.9, epsilon = 0.1, beta = 1, plot_heat=True, choose_acti
Q, state_visits, rewards, steps = sarsa(env, Q, state_visits, alpha=0.4, gamma=0.9, epsilon = 0.1, beta = 1, plot_heat=True, choose_actio
```

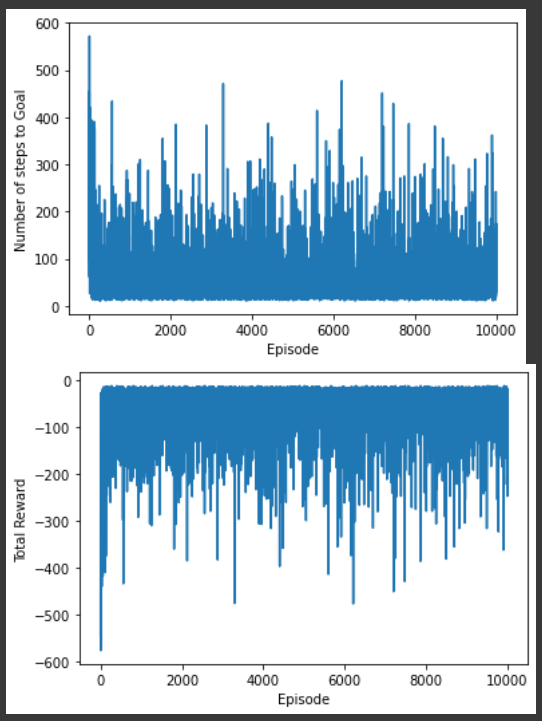





```
100%|██████████| 10000/10000 [01:54<00:00, 87.14it/s]
```

```
# TODO: visualize individual metrics vs episode count (averaged across multiple run(s))

plt.figure()
plt.plot(steps)
plt.xlabel('Episode')
```

```
plt.ylabel('Number of steps to Goal')
plt.show()

plt.figure()
plt.plot(rewards)
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.show()
```



Visualizing the policy

Now let's see the agent in action. Run the below cell (as many times) to render the policy;

Q-Learning

Now, implement the Q-Learning algorithm as an exercise.

Recall the update rule for Q-Learning:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Hyperparameters

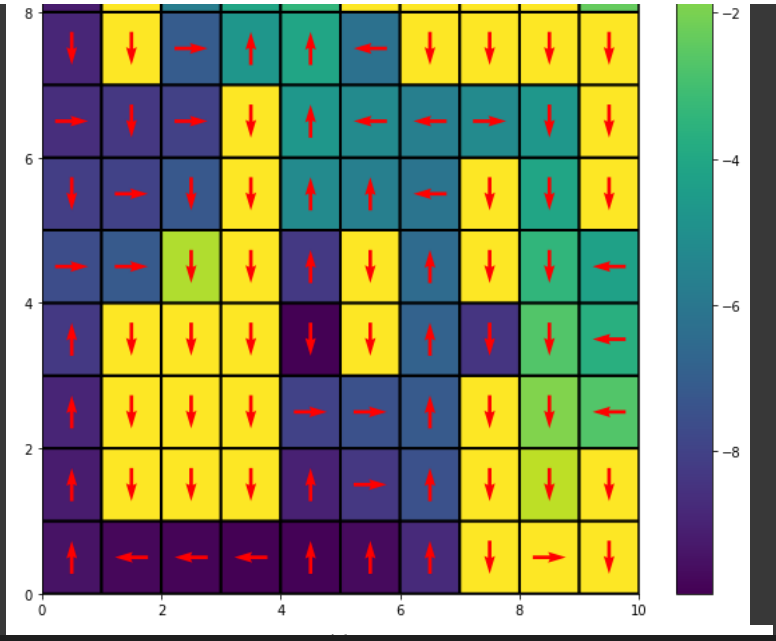
So we have some hyperparameters for the algorithm:

- α (learning rate)
- γ (discount factor)
- ϵ: For epsilon greedy exploration
- β: For Softmax exploration (temperature)

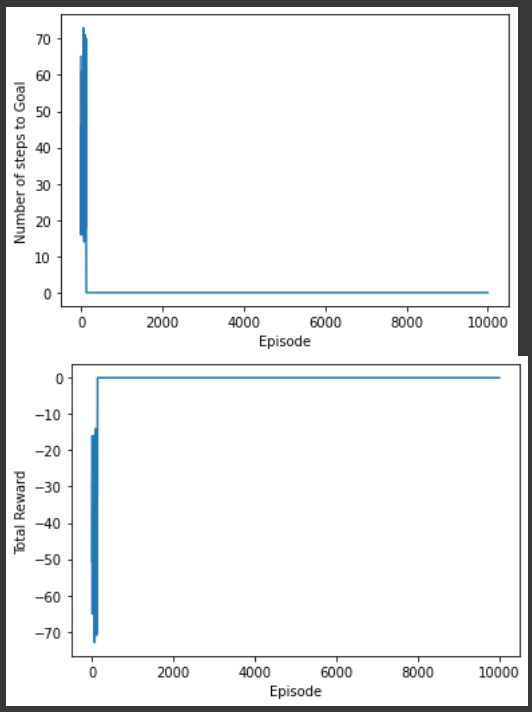
```
print_freq = 100

def qlearning(env, Q, state_visits, alpha=0.4, gamma=0.9, epsilon = 0.1, beta = 1, plot_heat=False, choose_action=choose_action_softmax):

    episode_rewards = np.zeros(episodes)
    steps_to_completion = np.zeros(episodes)
    if plot_heat:
        clear_output(wait=True)
        plot_Q(Q)
        plot_state_visits(state_visits)
    #epsilon = epsilon0
    #alpha = alpha0
    for ep in tqdm(range(episodes)):
        total_reward, steps = 0, 0

        # Reset environment
        state = env.reset()
```

```
        step_count = 0

        Q = np.reshape(Q,(100,4))
        state_visits = np.reshape(state_visits,(100))
        action = choose_action(Q, state, epsilon, beta)
        done = False
        while not done:
            step_count = step_count + 1
            state_next, reward = env.step(state,action)
            if (state_next in env.goal_states_seq):
                done = True
                break;
            action_next = choose_action(Q, state_next, epsilon, beta)

            # TODO: update equation
            max_next_Q = np.max(Q[state_next, :])
            Q[state,action] = Q[state,action] + alpha *(reward + gamma * max_next_Q - Q[state,action])
            state_visits[state] += 1;
            total_reward += reward
            steps += 1

            state, action = state_next, action_next
            if (step_count > 100):
                return Q, state_visits, episode_rewards, steps_to_completion

        episode_rewards[ep] = total_reward
        steps_to_completion[ep] = steps

        Q = np.reshape(Q,(10,10,4))
        state_visits = np.reshape(state_visits,(10,10))

        if (ep+1)%print_freq == 0 and plot_heat:
            clear_output(wait=True)
            plot_Q(Q, message = "Episode %d: Reward: %f, Steps: %.2f, Qmax: %.2f, Qmin: %.2f"%(ep+1, np.mean(episode_rewards[ep-print_fre
                                                                                np.mean(steps_to_completion[ep-print_freq+1:ep]),
                                                                                Q.max(), Q.min()))
            plot_state_visits(state_visits)

    return Q, state_visits, episode_rewards, steps_to_completion
```

```
#Q, state_visits, rewards, steps = qlearning(env, Q, state_visits, alpha=0.4, gamma=0.9, epsilon = 0.1, beta = 1, plot_heat=True, choose_
Q, state_visits, rewards, steps = qlearning(env, Q, state_visits, alpha=0.4, gamma=0.9, epsilon = 0.1, beta = 1, plot_heat=True, choose_a
```



```
# TODO: visualize individual metrics vs episode count (averaged across multiple run(s))
plt.figure()
plt.plot(steps)
plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.show()

plt.figure()
plt.plot(rewards)
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.show()
```



Visualizing the policy

Now let's see the agent in action. Run the below cell (as many times) to render the policy;

```
#from time import sleep

#state = env.reset()
#done = False
#steps = 0
#tot_reward = 0
#while not done:
#    clear_output(wait=True)
#    row_col_state = seq_to_col_row(state,10)
```

```
#    state, reward = env.step(state, Q[row_col_state[0], row_col_state[1]].argmax())
#    plt.figure(figsize=(10, 10))
#    env.render(ax=plt, render_agent=True)
#    plt.show()
#    steps += 1
#    tot_reward += reward
#    sleep(0.2)
#print("Steps: %d, Total Reward: %d"%(steps, tot_reward))
```

✓ 0s completed at 10:40 PM