Learning Based Control HW3: Reinforcement Learning

Submitted By: Ashutosh Gupta
OSU ID: 934533517

I worked on this assignment on my own. All the work below along with the attached code (except the grid-world code, which was provided with the problem) is my own work. To clarify some doubts I worked with Joshua Cook and Manuel Agraz.

Contents

1 SARSA Learning

1.1 Implementation

SARSA learning is an Reinforcement Learning value based policy that learns Q-values of the state-action pairs (s,a) using current state, current action, reward, next state and next action i.e. using (s,a,r,s',a') tuples. SARSA learning is policy dependent as it needs the next action from the policy to update the current Q-values. The update rule for SARSA is

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

Where α is the learning rate and γ is the discount factor. In our implementation of SARSA for the grid world problem, we zero initialize the Q-table. The policy chooses an action from a list of actions $[Up, Down, Left, Right, Stay]$, by the ε-greedy algorithm. Policy chooses a random action with probability of ε or chooses the action with the highest Q-value at the current state. The exploration factor (ε) is decayed at every episode by a decay factor (β). Each episode is run for a given maximum time steps and the agent receives a termination when it reaches the door (goal position). Below are the parameters for the implementation

- Episodes $e = 500$
- Time steps $t = 200$
- Learning rate $\alpha = 0.25$
- Discount factor $\gamma = 0.9$
- Exploration rate $\varepsilon = 0.9$
- Exploration Decay rate $\beta = 0.99$

1.2 Results

The learning curve - rewards against episodes is plotted in Figure 1a. Q-values for each state and the four moving actions (except stay) is shown as heat maps in Figure 1b. The highest Q-value for each state in the grid is then plotted as arrows for the corresponding action in Figure 1c, the black square depicts the "Stay" action.

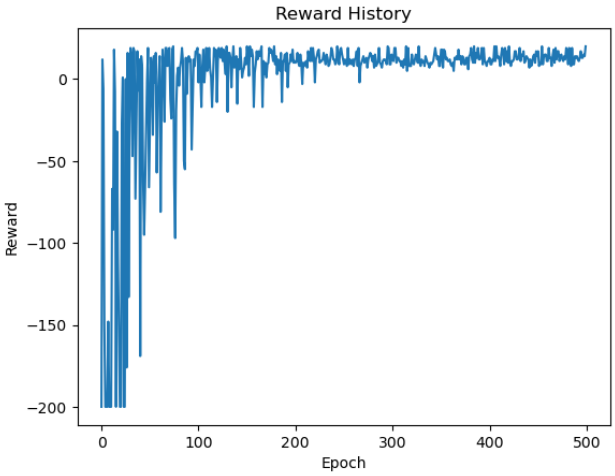
The learning does converge to a good policy to learn to get to the goal position. Although there is more randomness in the starting episodes and it takes the policy longer to converge to good rewards.

2 Q Learning

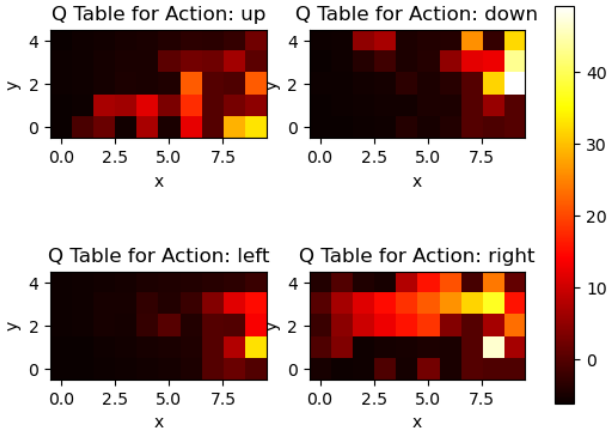
2.1 Implementation

Q-learning is an Reinforcement Learning value based policy that learns Q-values of the state-action pairs (s,a) using current state, current action, reward, next state and maximum of Q-value for next state i.e. using (s,a,r,s',max_a) tuples. Q-learning is policy independent as it uses the maximum of the Q-value for the next state to update the current Q-values instead of using the policy for the next action. Due to this Q-learning removes the randomness in the update caused by exploration by the policy as it always uses the best Q-value. The update rule for Q-learning is

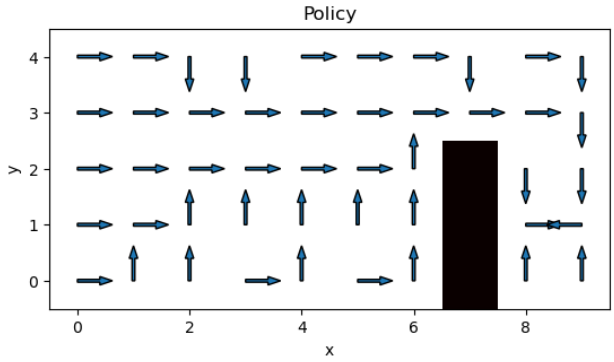
$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha * [r(s_t,a_t) + \gamma * max_a Q(s_{t+1},a) - Q(s_t,a_t)] \tag{2}$$



(a) Reward vs Episodes for SARSA learning



(b) Q-values for each state and each action as heat map



(c) Highest Q-value for each state plotted as arrows

Figure 1: SARSA learning results for fixed goal

Where α is the learning rate and γ is the discount factor. In our implementation of Q-learning for the grid world problem, we zero initialize the Q-table. The policy chooses an action from a list of actions $[Up, Down, Left, Right, Stay]$, by the ε-greedy algorithm. Policy chooses a random action with probability of ε or chooses the action with the highest Q-value at the current state. The exploration factor (ε) is decayed at every episode by a decay factor (β). Each episode is run for a given maximum time steps and the agent receives a termination when it reaches the door (goal position). Below are the parameters for the implementation

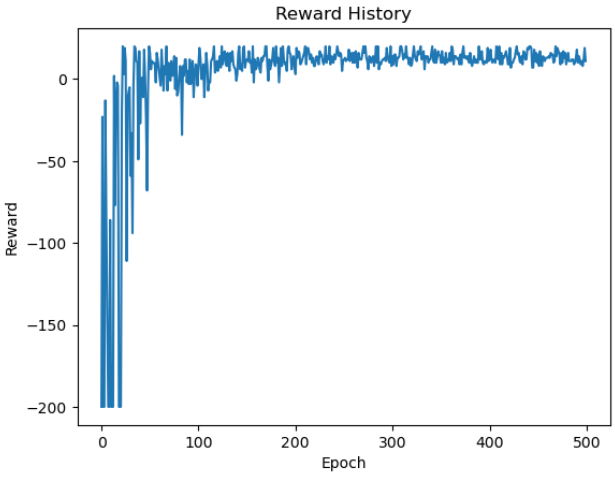
- Episodes $e = 500$
- Time steps $t = 200$
- Learning rate $\alpha = 0.25$
- Discount factor $\gamma = 0.9$
- Exploration rate $\varepsilon = 0.9$
- Exploration Decay rate $\beta = 0.99$

The only difference in SARSA (Equation 1) and Q-learning (Equation 2) implementation is the respective update rules. The parameters and choosing action is all similar.

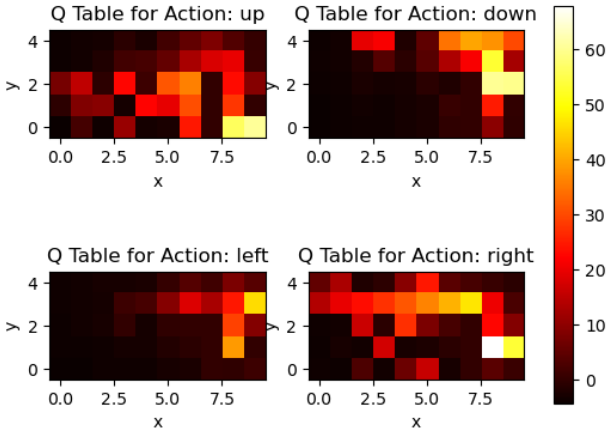
2.2 Results

The learning curve - rewards against episodes is plotted in Figure 2a. Q-values for each state and the four moving actions (except stay) is shown as heat maps in Figure 2b. The highest Q-value for each state in the grid is then plotted as arrows for the corresponding action in Figure 2c, the black square depicts the "Stay" action.

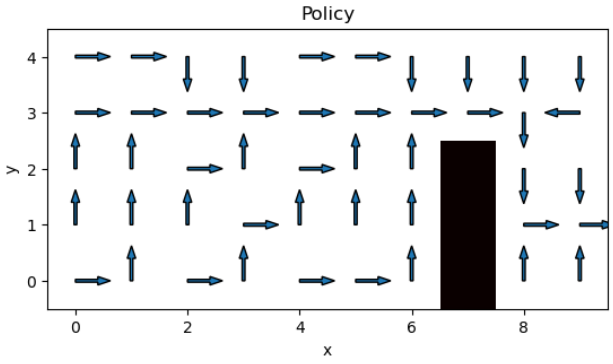
The learning does converge to a good policy to learn to get to the goal position. There is lesser randomness in the starting episodes and it takes the policy shorter to converge to good rewards as compared to SARSA. The Q-policy also converges better as evident by the reward curves and the Q-table.



(a) Reward vs Episodes for Q-learning



(b) Q-values for each state and each action as heat map



(c) Highest Q-value for each state plotted as arrows

Figure 2: Q-learning results for fixed goal

3 Comparison SARSA vs Q-learning

A comparison of SARSA and Q-learning reward curves is shown in Figure 3. It shows that Q-learning converges faster than SARSA and it also has less deviation after convergence. This is an expected result as Q-learning is a more optimal policy compared to SARSA because Q-learning only uses the best Q-value at each state to update the table. This removes the bad values caused by policy exploration to affect the Q-values, which results in better learning. Q-learning is therefore theoretically expected to converge to an optimal policy given enough number of episodes, while SARSA is not expected to get to the optimal policy.

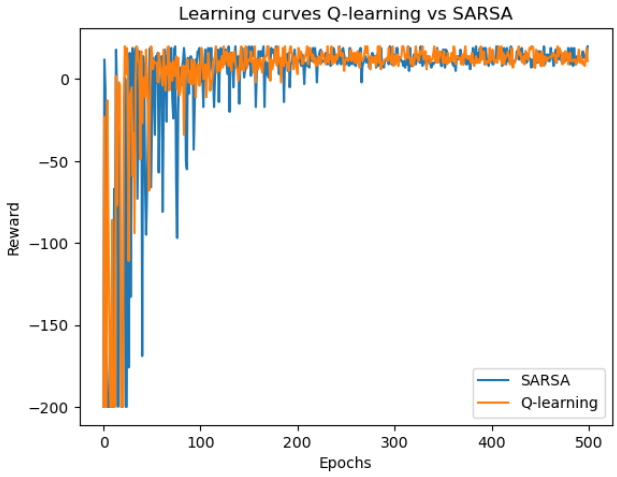


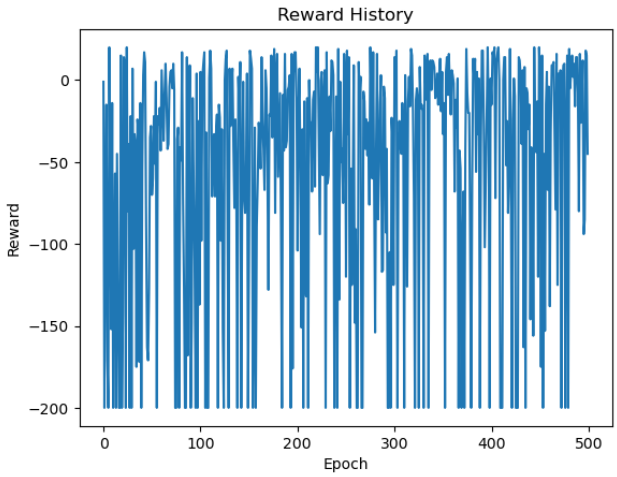
Figure 3: Reward curve for SARSA vs Q-learning for fixed goal

4 Moving Goal

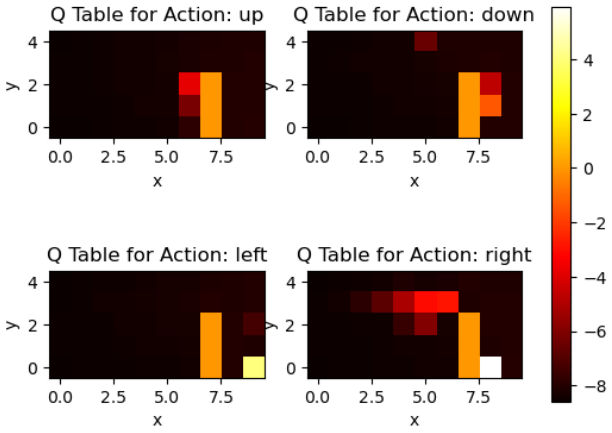
All the above results were shown for a fixed goal position. That means there was no randomness in the dynamics of the environment as we had a fixed goal. Now keeping all the above parameters same for both the policies we randomly shift the door by one cell at every time step. Reward curve, learned Q-table and the policy for SARSA learning with a moving target is shown in Figure 4. Similar plots for the Q-learning are shown in Figure 5. And a comparison of reward curves for SARSA and Q-learning for a moving goal is shown in Figure 6.

As it is evident by the learning curves and the Q-tables the policies for both cases - SARSA and Q-learning do not converge to anything. As seen in the reward curves agents receive good rewards sometimes, but that could be due to the randomness in the initial spawning really close to the goal. Both the agents perform really poorly as randomization is introduced in the dynamics of the environment. This simple implementation of SARSA and Q-learning cannot handle dynamics randomness that well and so do not learn a good policy. Both policies face the exploration challenge,where policy needs to explore more to understand goal behavior. Due to randomness in the goal movement, both policies cannot predict the behavior and so an optimal policy changes every time step to reach the goal.

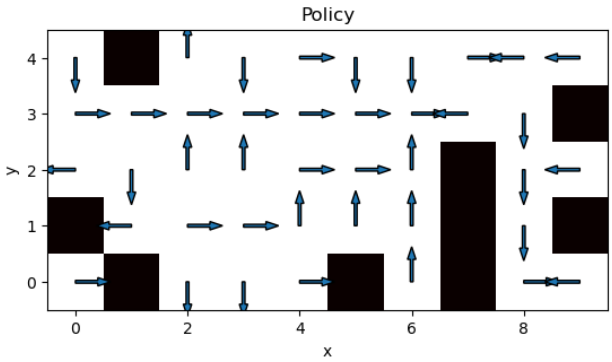
To improve performance we could introduce function approximations to predict the goal behavior. This could range from learning a model for the environment separately and using SARSA or Q-learning with that model. Or we could also directly incorporate learning in the policy as Deep Q-learning.



(a) Reward vs Episodes for SARSA learning

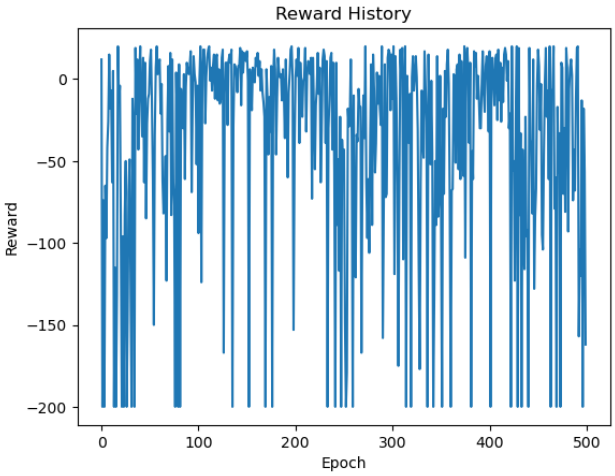


(b) Q-values for each state and each action as heat map

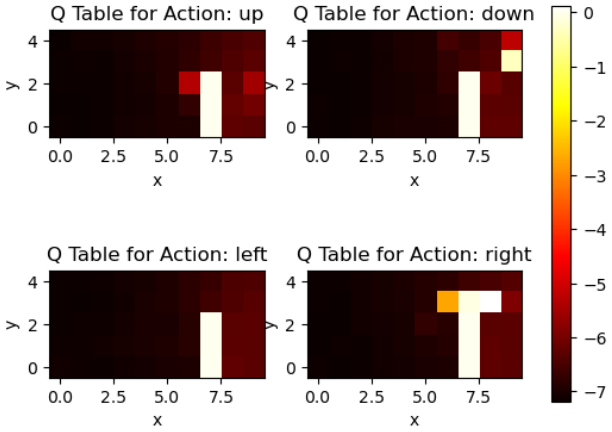


(c) Highest Q-value for each state plotted as arrows

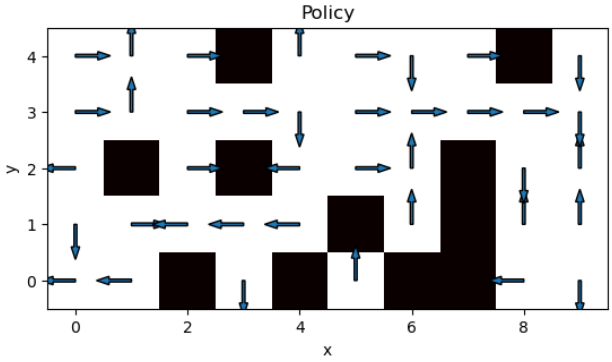
Figure 4: SARSA learning results for moving goal



(a) Reward vs Episodes for Q-learning



(b) Q-values for each state and each action as heat map



(c) Highest Q-value for each state plotted as arrows

Figure 5: Q-learning results for moving goal

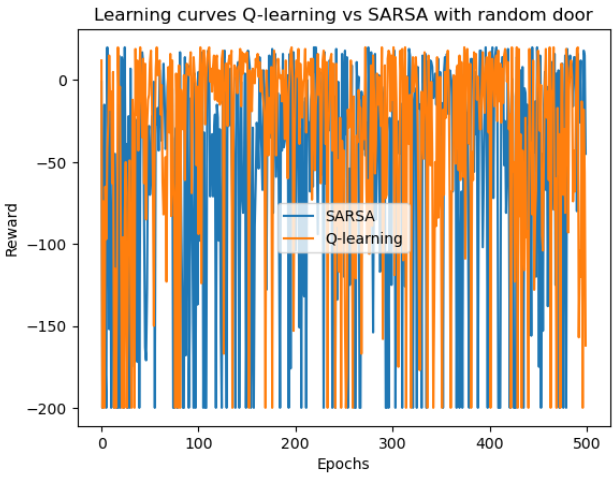


Figure 6: Reward curve SARSA vs Q-learning for moving goal

A Learning algorithm code

A.1 SARSA Learning

```
import numpy as np
import matplotlib.pyplot as plt

from gridWorld import gridWorld


class sarsaLearner(gridWorld):
    """SARSA learning for the gridWorld environment of 10x5 cells
    Learns at every step from (s,a,r,s',a') tuples
    """

    def __init__(self, epochs:int, time_steps:int, alpha:float, gamma:float,
    ↪  epsilon:float, epsilon_decay:float, rng_door:bool=False):
        """Initialize the SARSA learner

        Args:
            epochs (int): number of episodes to run
            time_steps (int): number of time steps per episode
            alpha (float): learning rate
            gamma (float): discount factor
            epsilon (float): exploration rate
            epsilon_decay (float): exploration rate decay factor
            rng_door (bool, optional): whether to randomly move the door. Defaults
            ↪  to False.
        """

        # Initialize the gridWorld environment class
        super().__init__()

        # Parameters
        self.epochs = epochs
        self.time_steps = time_steps
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.rng_door = rng_door

        # Actions
        self.actions = ["up", "down", "left", "right", "stay"]

        # Initialize the reward history
        self.reward_history = []
```

```
        # Initialize Q table as dictionary of dictionaries
        # Dictonaries are the fastest way to access the Q values
        # Dictionary keys are states, values are dictionaries of actions and Q
        ↪ values
        self.Q_table = {}
        for x in range(10):
            for y in range(5):
                self.Q_table[(x, y)] = {}
                for action in self.actions:
                    # Initialize Q values to 0
                    self.Q_table[(x, y)][action] = 0

    def _choose_action(self, state:tuple):
        """Choose an action based on the epsilon greedy policy for exploration

        Args:
            state (tuple): current state
        """

        # Explore with probability epsilon
        if np.random.rand() < self.epsilon:
            action = np.random.choice(self.actions)

        # Exploit with probability 1-epsilon
        else:
            # Get the Q values for the current state
            Q_values = self.Q_table[state]
            # Take the action with the max Q value
            maxQ = max(Q_values.values())
            # Random choice if multiple actions have the same max Q value
            action = np.random.choice([k for k,v in Q_values.items() if v==maxQ])

        return action

    def _learn_sarsa(self, s:tuple, a:str, r:int, s_prime:tuple, a_prime:str):
        """Update the Q values based on the SARSA update rule

        Args:
            s (tuple): current state
            a (str): current action
            r (int): reward
            s_prime (tuple): next state
            a_prime (str): next action
        """
```

```
        # Get the Q values for the current state and action
        Q_sa = self.Q_table[s][a]
        # Get the Q values for the next state and action
        Q_sa_prime = self.Q_table[s_prime][a_prime]

        # Update the Q value for the current state and action
        Q_sa = Q_sa + self.alpha*(r + self.gamma*Q_sa_prime - Q_sa)
        # Update the Q table
        self.Q_table[s][a] = Q_sa

    def run(self, verbose:bool=False):
        """Run the SARSA learner

        Args:
            verbose (bool, optional): whether to print progress. Defaults to
             ↪  False.
        """

        # Run for the specified number of epochs
        for epoch in range(self.epochs):

            # Reset the environment
            current_state = tuple(self.reset())

            # Initialize the reward for this epoch
            epoch_reward_history = 0

            # Choose an initial action based on the epsilon greedy policy
            action = self._choose_action(current_state)

            # Run for the specified number of time steps
            for time_step in range(self.time_steps):

                # Take the action and get the next state and reward
                next_state, reward = self.step(action, self.rng_door)
                next_state = tuple(next_state)

                # Choose next action based on the epsilon greedy policy
                next_action = self._choose_action(next_state)

                # Update the Q values based on the SARSA update rule
                self._learn_sarsa(current_state, action, reward, next_state,
                 ↪  next_action)

                # Update the current state
                current_state = next_state
```

```
            # Update the next action
            action = next_action

            # Update the reward for this epoch
            epoch_reward_history += reward

            # If the agent reaches the door, break
            if reward == 20:
                break

        # Update the reward history
        self.reward_history.append(epoch_reward_history)

        # Decay the exploration rate
        self.epsilon *= self.epsilon_decay

        # If verbose, print the progress
        if verbose and (epoch%100==0 or epoch==self.epochs-1):
            print(f"Epoch: {epoch+1}/{self.epochs} | Reward for epoch:
             ↪ {epoch_reward_history}")

        # If verbose, print the Q_values for the door
        if verbose and epoch==self.epochs-1 and not self.rng_door:
            print(f"Q Values for Door: {self.Q_table[tuple(self.init_door)]}")

def plot_reward_history(self):
    """Plot the reward history"""

    # Plot the reward history
    plt.figure("Reward History")
    plt.plot(self.reward_history)
    plt.title("Reward History")
    plt.xlabel("Epoch")
    plt.ylabel("Reward")

def plot_Q_table(self):
    """Plot the Q table as heatmap for each state and action"""

    # Initialize the Q table
    Q_table = np.zeros((5, 10, 5))

    # Fill in the Q table
    for x in range(10):
        for y in range(5):
            Q_values = self.Q_table[(x, y)]
```

```
                for i, action in enumerate(self.actions):
                    Q_table[y, x, i] = Q_values[action]

        # Plot the Q table for each action except stay
        plt.figure("Q Table")
        for i, action in enumerate(self.actions[:-1]):
            plt.subplot(2, 2, i+1)
            plt.imshow(Q_table[:, :, i], cmap="hot", vmin=np.min(Q_table),
             ↪  vmax=np.max(Q_table))
            plt.gca().invert_yaxis()
            plt.xlabel("x")
            plt.ylabel("y")
            plt.title(f"Q Table for Action: {action}")

        # Plot a single colorbar
        plt.colorbar(ax=plt.gcf().get_axes(), shrink=0.9, location="right")

    def plot_policy(self):
        """Plot the policy as arrows pointing in the direction of the action"""

        plt.figure("Policy")

        # Initialize the policy
        policy = np.zeros((5, 10))

        # Fill in the policy
        for x in range(10):
            for y in range(5):
                Q_values = self.Q_table[(x, y)]
                maxQ = max(Q_values.values())
                action = np.random.choice([k for k,v in Q_values.items() if
                 ↪  v==maxQ])

                # For wall cells, just plot stay as the Q_values are all 0 and
                 ↪  random.choice just picking randomly
                if x==7 and y<3:
                    action = "stay"

                # Assign a 1 if action is not stay
                # Plot corresponding arrows
                if action == "up":
                    policy[y, x] = 1
                    plt.arrow(x, y, 0, 0.4, width=0.05)
                elif action == "down":
                    policy[y, x] = 1
                    plt.arrow(x, y, 0, -0.4, width=0.05)
```

```
                elif action == "right":
                    policy[y, x] = 1
                    plt.arrow(x, y, 0.4, 0, width=0.05)
                elif action == "left":
                    policy[y, x] = 1
                    plt.arrow(x, y, -0.4, 0, width=0.05)
                elif action == "stay":
                    # Just plot stay as different color
                    policy[y, x] = 0

        # Plot the policy
        plt.imshow(policy, cmap="hot", interpolation="nearest")
        plt.gca().invert_yaxis()
        plt.xlabel("x")
        plt.ylabel("y")
        plt.title("Policy")


if __name__ == "__main__":

    # Parameters
    epochs = 500
    time_steps = 200
    alpha = 0.25 # Learning rate
    gamma = 0.9 # Discount factor
    epsilon = 0.9 # Exploration factor
    epsilon_decay = 0.99 # Exploration decay factor
    rng_door = False # Move the door randomly

    # Initialize the SARSA learner
    sarsaAgent = sarsaLearner(epochs, time_steps, alpha, gamma, epsilon,
    ↪  epsilon_decay, rng_door)

    # Run the SARSA learner
    sarsaAgent.run(verbose=True)

    # Generate plots
    sarsaAgent.plot_reward_history()
    sarsaAgent.plot_Q_table()
    sarsaAgent.plot_policy()

    # Show the plots
    plt.show()
```

A.2 Q Learning

```python
import numpy as np
import matplotlib.pyplot as plt

from sarsaLearner import sarsaLearner


class qLearner(sarsaLearner):
    """Q learning for the gridWorld environment of 10x5 cells
    Learns at every step from (s,a,r,s') tuples
    """

    def __init__(self, epochs:int, time_steps:int, alpha:float, gamma:float,
    ↪  epsilon:float, epsilon_decay:float, rng_door:bool=False):
        """Initialize the Q learner

        Args:
            epochs (int): number of episodes to run
            time_steps (int): number of time steps per episode
            alpha (float): learning rate
            gamma (float): discount factor
            epsilon (float): exploration rate
            epsilon_decay (float): exploration rate decay factor
            rng_door (bool, optional): whether to randomly move the door. Defaults
            ↪  to False.
        """

        # Class derived from sarsaLearner, as only the _learn_xx and run methods
        ↪  need to be changed
        # _choose_action and plot_xx methods are the same as sarsaLearner
        # Initialize the SARSA learner class
        super().__init__(epochs, time_steps, alpha, gamma, epsilon, epsilon_decay,
        ↪  rng_door)

    def _learn_q(self, s:tuple, a:str, r:float, s_prime:tuple):
        """Update the Q values based on the Q learning update rule

        Args:
            s (tuple): current state
            a (str): action taken
            r (float): reward received
            s_prime (tuple): next state
        """

        # Get Q values for current state and action
        Q_sa = self.Q_table[s][a]
```

```
        # Get Q values for next state
        Q_s_prime = self.Q_table[s_prime]

        # The only difference between SARSA and Q learning is that Q learning uses
        ↪  the maximum Q value for the next state
        # SARSA uses the Q value for the next state and next action
        # Get the maximum Q value for the next state
        max_Q_s_prime = max(Q_s_prime.values())

        # Update the Q value for the current state and action
        Q_sa = Q_sa + self.alpha * (r + self.gamma * max_Q_s_prime - Q_sa)
        # Update the Q table
        self.Q_table[s][a] = Q_sa

    def run(self, verbose:bool=False):
        """Run the Q learning algorithm

        Args:
            verbose (bool, optional): whether to print progress. Defaults to
            ↪  False.
        """

        # Run for the specified number of epochs
        for epoch in range(self.epochs):

            # Reset the environment
            current_state = tuple(self.reset())

            # Initialize the reward for this epoch
            epoch_reward_history = 0

            # Run for the specified number of time steps
            for time_step in range(self.time_steps):

                # Choose an action based on the epsilon greedy policy
                action = self._choose_action(current_state)

                # Take the action and get the next state and reward
                next_state, reward = self.step(action, self.rng_door)
                next_state = tuple(next_state)

                # Update the Q values based on the Q learning update rule
                self._learn_q(current_state, action, reward, next_state)

                # Update the current state
```

```
                current_state = next_state

                # Update the reward for this epoch
                epoch_reward_history += reward

                # If the agent reaches the door, break
                if reward == 20:
                    break

            # Update the reward history
            self.reward_history.append(epoch_reward_history)

            # Decay the exploration rate
            self.epsilon *= self.epsilon_decay

            # If verbose, print the progress
            if verbose and (epoch%100==0 or epoch==self.epochs-1):
                print(f"Epoch {epoch+1}/{self.epochs} | Reward for epoch:
                 ↪ {epoch_reward_history}")

            # If verbose, print the Q_values for the door
            if verbose and epoch==self.epochs-1 and not self.rng_door:
                print(f"Q Values for Door: {self.Q_table[tuple(self.init_door)]}")


if __name__=="__main__":

    # Parameters
    epochs = 500
    time_steps = 200
    alpha = 0.25 # Learning rate
    gamma = 0.9 # Discount factor
    epsilon = 0.9 # Exploration factor
    epsilon_decay = 0.99 # Exploration decay factor
    rng_door = False # Move the door randomly

    # Initialize the Q learner
    qAgent = qLearner(epochs, time_steps, alpha, gamma, epsilon, epsilon_decay,
     ↪ rng_door)

    # Run the Q learner
    qAgent.run(verbose=True)

    # Generate plots
    qAgent.plot_reward_history()
    qAgent.plot_Q_table()
```

```
    qAgent.plot_policy()

    # Show the plots
    plt.show()
```

A.3 Grid World code

```
from random import randint
from copy import deepcopy

class gridWorld:
    def __init__(self):
        self.init_door=[9,1]
        self.agent=None
        self.door=None

    def reset(self):
        self.door=self.init_door
        position=[-1,-1]
        while not self.isValid(position):
            position[0]=randint(0,9)
            position[1]=randint(0,4)
        self.agent=position
        return deepcopy(self.agent)

    def take_action(self,action,position):
        x,y=position
        if action=="up":
            y+=1
        if action=="down":
            y-=1
        if action=="right":
            x+=1
        if action=="left":
            x-=1
        if self.isValid([x,y]):
            return [x,y]
        return position

    def step(self,action,rng_door=False):
        self.agent=self.take_action(action,self.agent)
        if rng_door:
            rng_action=["up","down","left","right"][randint(0,3)]
            self.door=self.take_action(rng_action,self.door)
        if self.agent[0]==self.door[0] and self.agent[1]==self.door[1]:
            reward=20
        else:
```

```
            reward=-1
        return deepcopy(self.agent),reward

    def isValid(self,position):
        x,y=position
        if x<0 or x>9:      #out of x bounds
            return False
        if y<0 or y>4:      #out of y bounds
            return False
        if x==7 and y<3:    #if wall
            return False
        return True


if __name__=="__main__":
    #example usage for a gym-like environment
    #state: [x,y] coordinate of the agent
    #actions: ["up","down","left","right"] directions the agent can move
    env=gridWorld()
    for learning_epoch in range(100):
        state=env.reset()                           #every episode, reset the
        ↪ environment to the original configuration
        for time_step in range(20):
            action=["up","down","left","right"][0] #learner chooses one of these
            ↪ actions
            next_state,reward=env.step(action)  #the action is taken, a reward and
            ↪ new state is returned
            #note: use env.step(action,rng_door=True) for part 2
```

B Experiments code

The pdf generated from the Jupyter notebook is attached below.

experiments

November 7, 2023

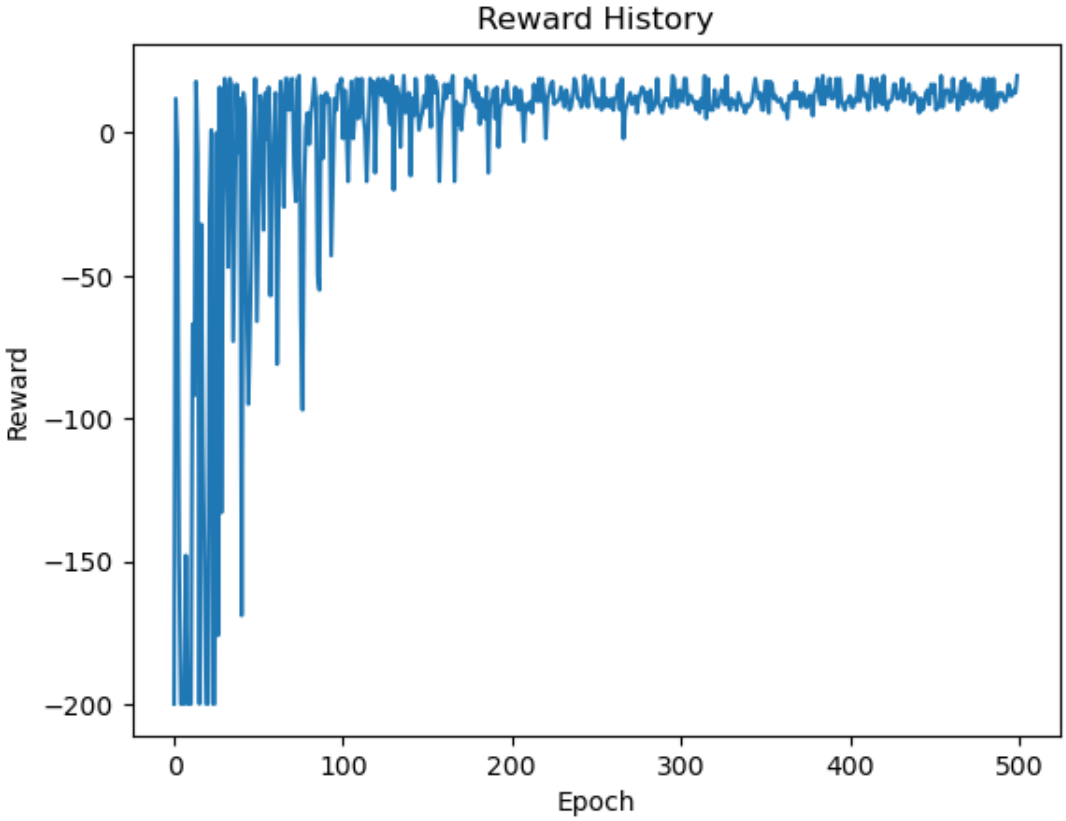
[1]:
```
import matplotlib.pyplot as plt

from sarsaLearner import sarsaLearner
from qLearner import qLearner
```

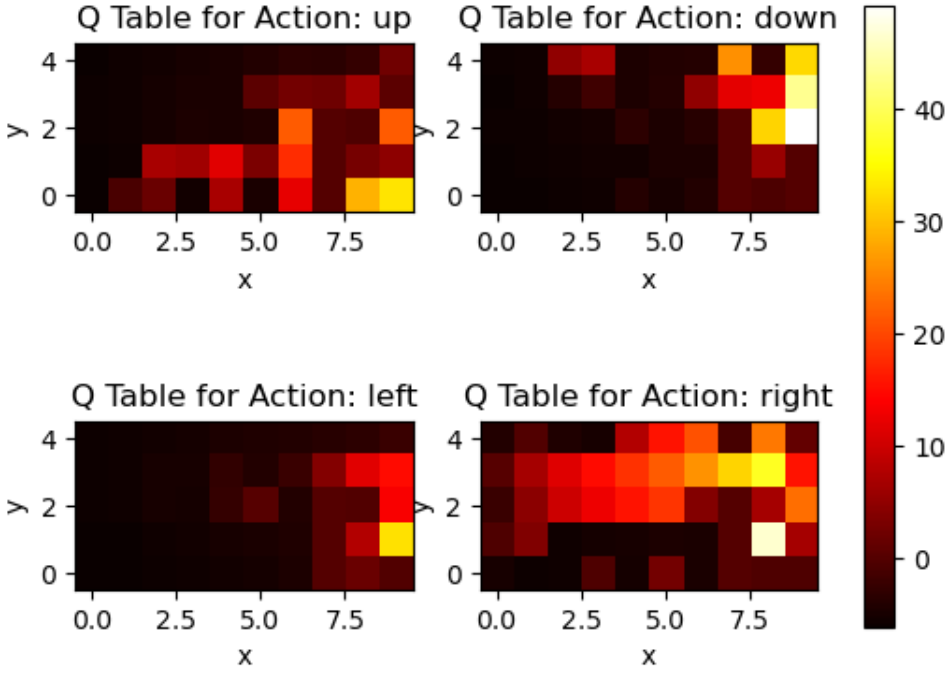
[2]:
```
# Common Parameters
epochs = 500 # number of episodes
time_steps = 200 # max time steps per episode
alpha = 0.25 # learning rate
gamma = 0.9 # discount factor
epsilon = 0.9 # exploration rate
epsilon_decay = 0.99 # exploration decay rate
```

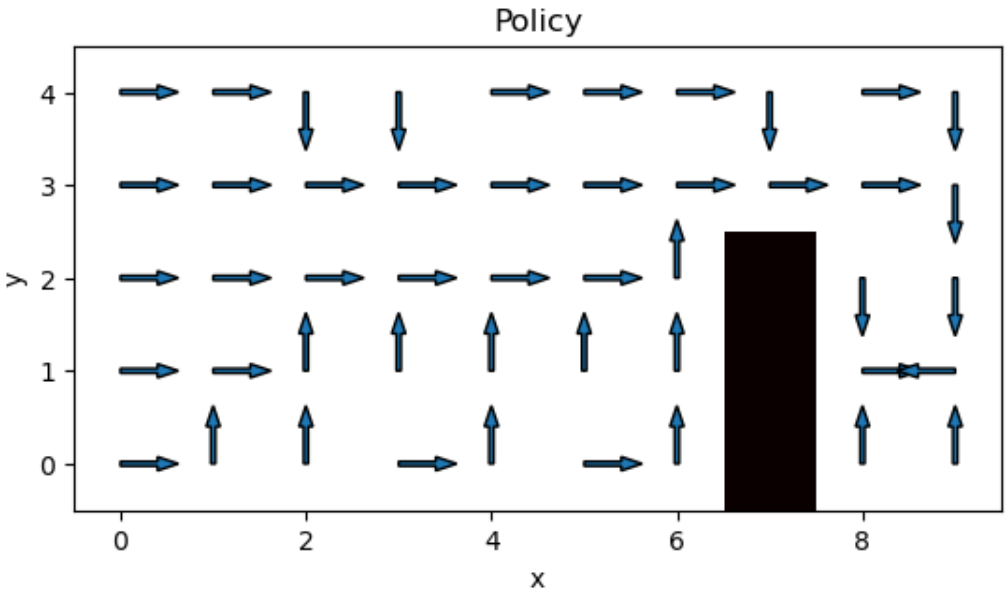
[9]:
```
## Experiments for SARSA learning

rng_door = False # random door

# Initialize learner
sarsaAgent = sarsaLearner(epochs, time_steps, alpha, gamma, epsilon,␣
 ↪epsilon_decay, rng_door)

# Run experiment
sarsaAgent.run(verbose=True)

# Generate plots
sarsaAgent.plot_reward_history()
sarsaAgent.plot_Q_table()
sarsaAgent.plot_policy()

# Show plots
plt.show()
```

```
Epoch: 1/500 | Reward for epoch: -200
Epoch: 101/500 | Reward for epoch: -2
Epoch: 201/500 | Reward for epoch: 10
Epoch: 301/500 | Reward for epoch: 12
Epoch: 401/500 | Reward for epoch: 13
Epoch: 500/500 | Reward for epoch: 20
```

```
Q Values for Door: {'up': 4.643981201075436, 'down': 0, 'left': 32.545240753983,
'right': 6.62335587469129, 'stay': 0}
```



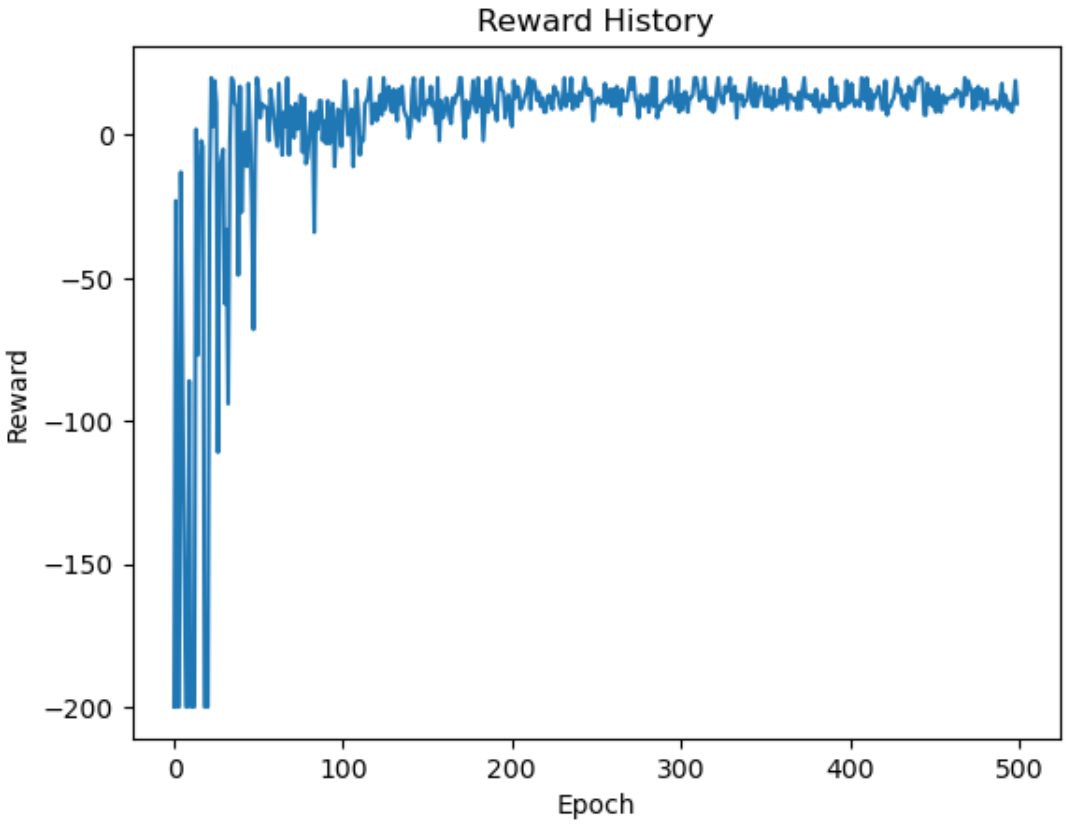


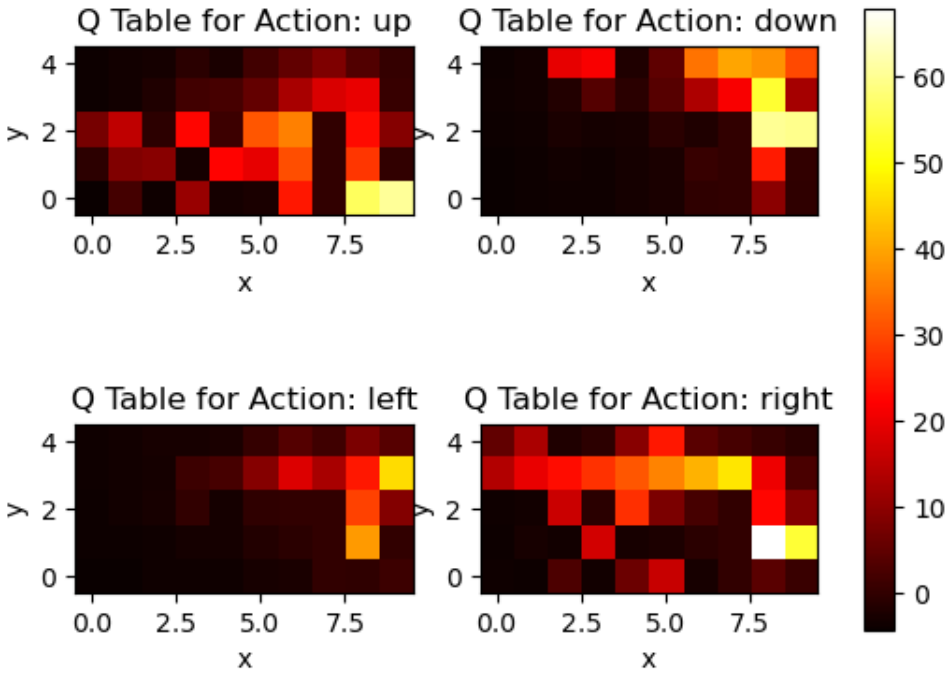


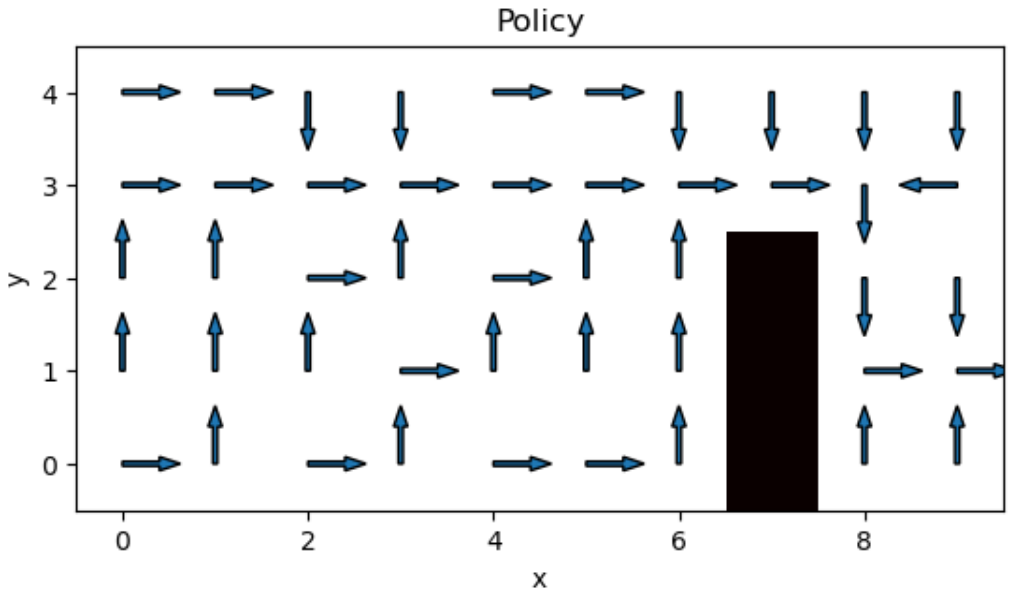
[12]:
```
## Experiments for Q-learning

rng_door = False # random door
```

```
# Initialize learner
qAgent = qLearner(epochs, time_steps, alpha, gamma, epsilon, epsilon_decay,␣
 ↪rng_door)

# Run experiment
qAgent.run(verbose=True)

# Generate plots
qAgent.plot_reward_history()
qAgent.plot_Q_table()
qAgent.plot_policy()

# Show plots
plt.show()
```

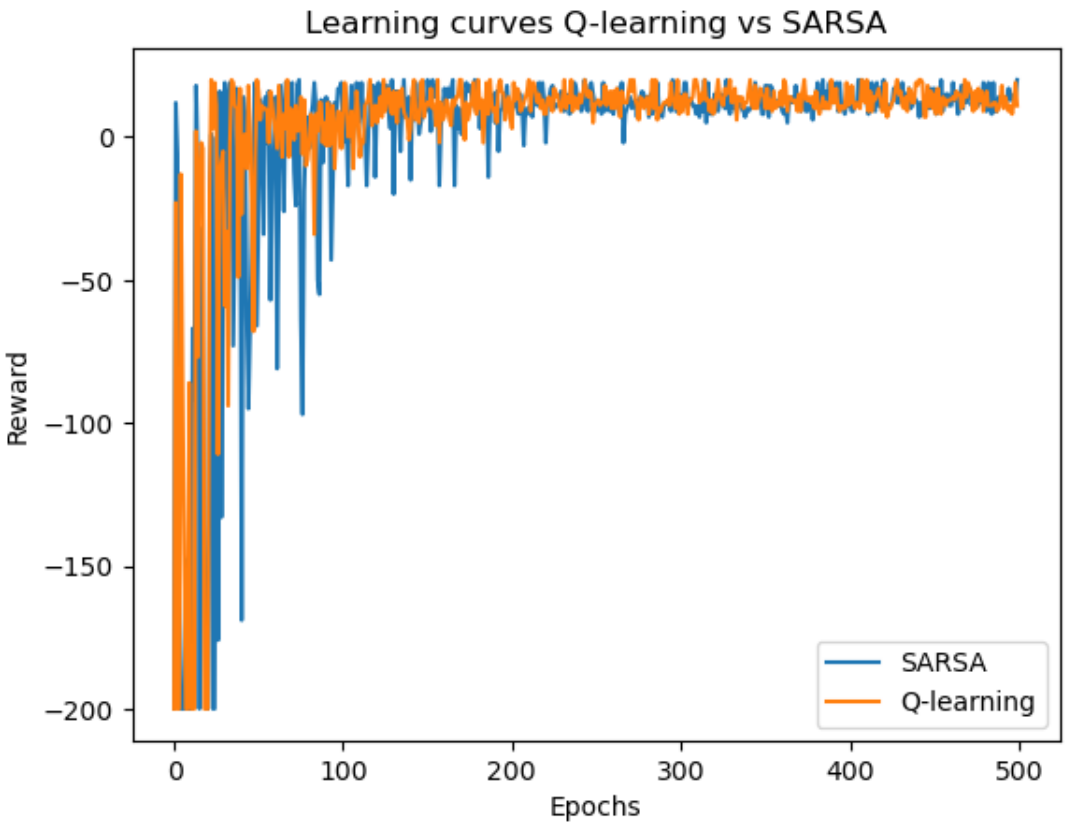
```
Epoch 1/500 | Reward for epoch: -200
Epoch 101/500 | Reward for epoch: 9
Epoch 201/500 | Reward for epoch: 3
Epoch 301/500 | Reward for epoch: 14
Epoch 401/500 | Reward for epoch: 12
Epoch 500/500 | Reward for epoch: 11
Q Values for Door: {'up': 0, 'down': 0, 'left': 0, 'right': 53.25186738754666,
'stay': 5.0}
```


Reward History
Reward
Epoch
0
−50
−100
−150
−200
0
100
200
300
400
500


Q Table for Action: up
Q Table for Action: down
Q Table for Action: left
Q Table for Action: right
x
y
0.0
2.5
5.0
7.5
0
2
4
0
10
20
30
40
50
60



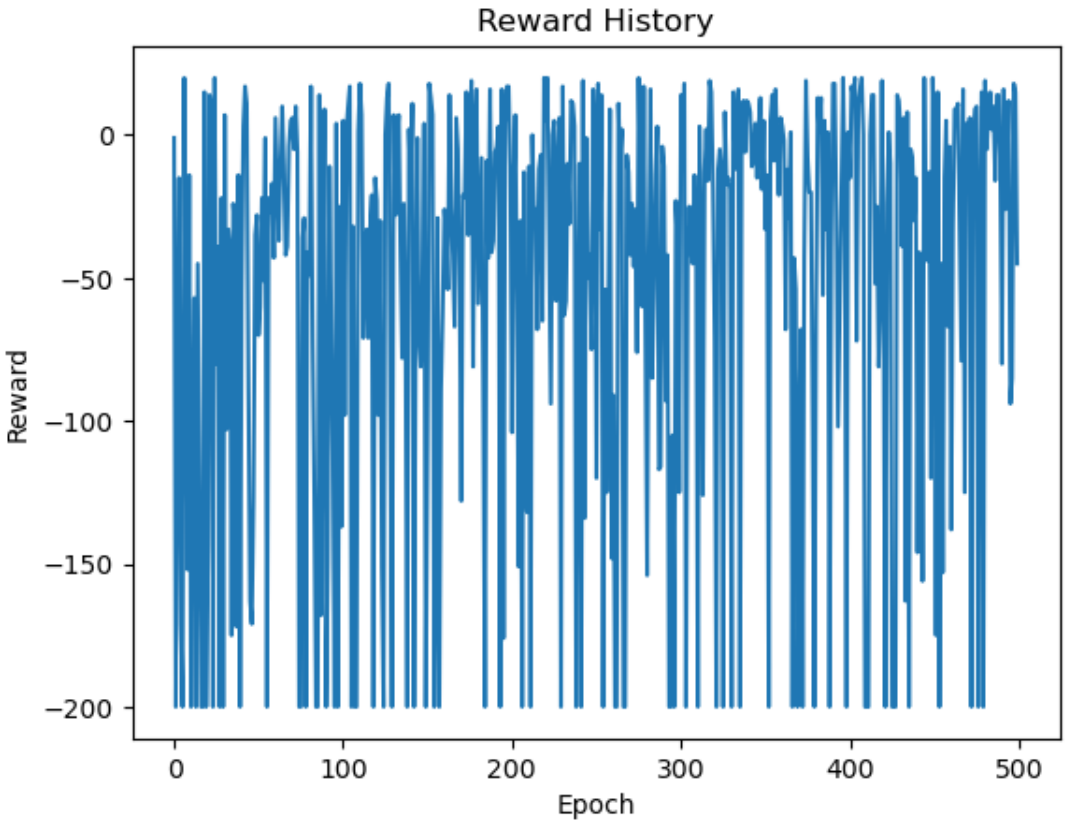
[13]:
```
# Overlay learning curves
plt.figure("Learning curves")
plt.plot(sarsaAgent.reward_history, label="SARSA")
plt.plot(qAgent.reward_history, label="Q-learning")
plt.xlabel("Epochs")
plt.ylabel("Reward")
plt.title("Learning curves Q-learning vs SARSA")
plt.legend()
plt.show()
```

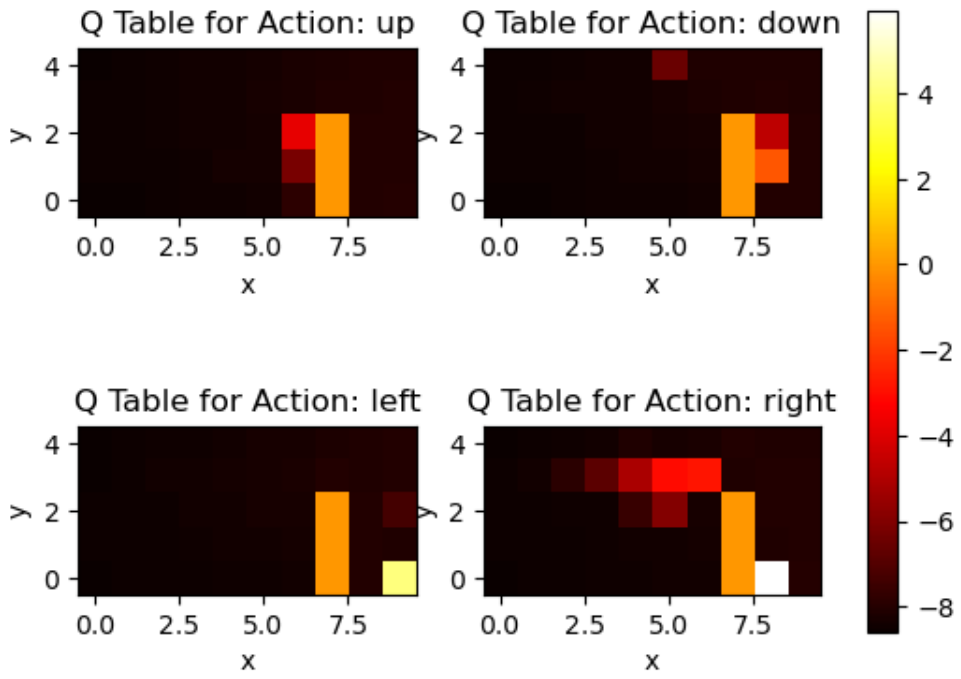


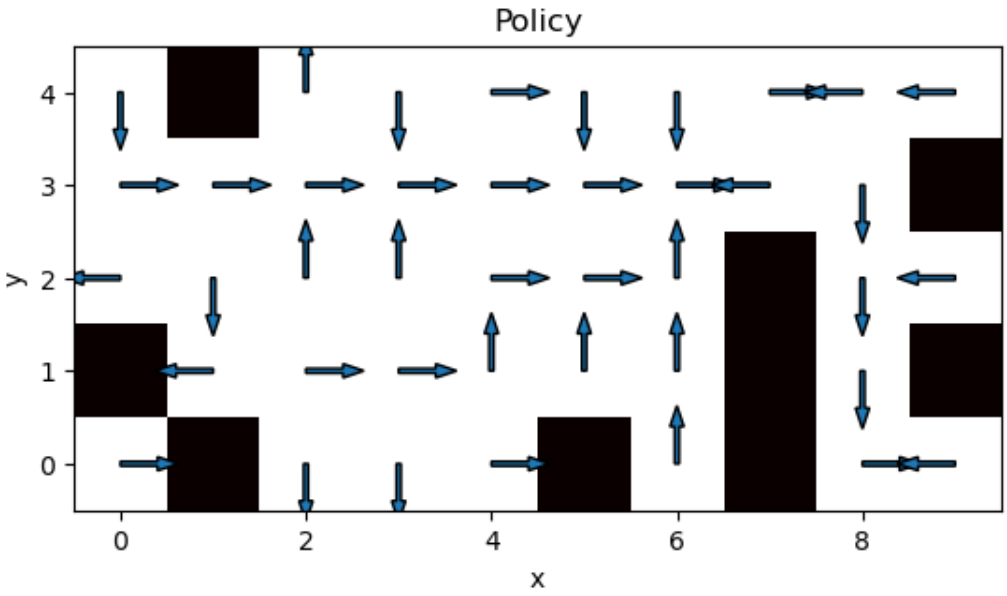
[22]:
```
## Experiments for SARSA learning with random door

rng_door = True # random door

# Initialize learner
sarsaAgent_rng_door = sarsaLearner(epochs, time_steps, alpha, gamma, epsilon,␣
 ↪epsilon_decay, rng_door)

# Run experiment
sarsaAgent_rng_door.run(verbose=True)

# Generate plots
sarsaAgent_rng_door.plot_reward_history()
sarsaAgent_rng_door.plot_Q_table()
sarsaAgent_rng_door.plot_policy()

# Show plots
plt.show()
```

```
Epoch: 1/500 | Reward for epoch: -1
```

```
Epoch: 101/500 | Reward for epoch: 5
Epoch: 201/500 | Reward for epoch: -104
Epoch: 301/500 | Reward for epoch: 14
Epoch: 401/500 | Reward for epoch: -15
Epoch: 500/500 | Reward for epoch: -45
```



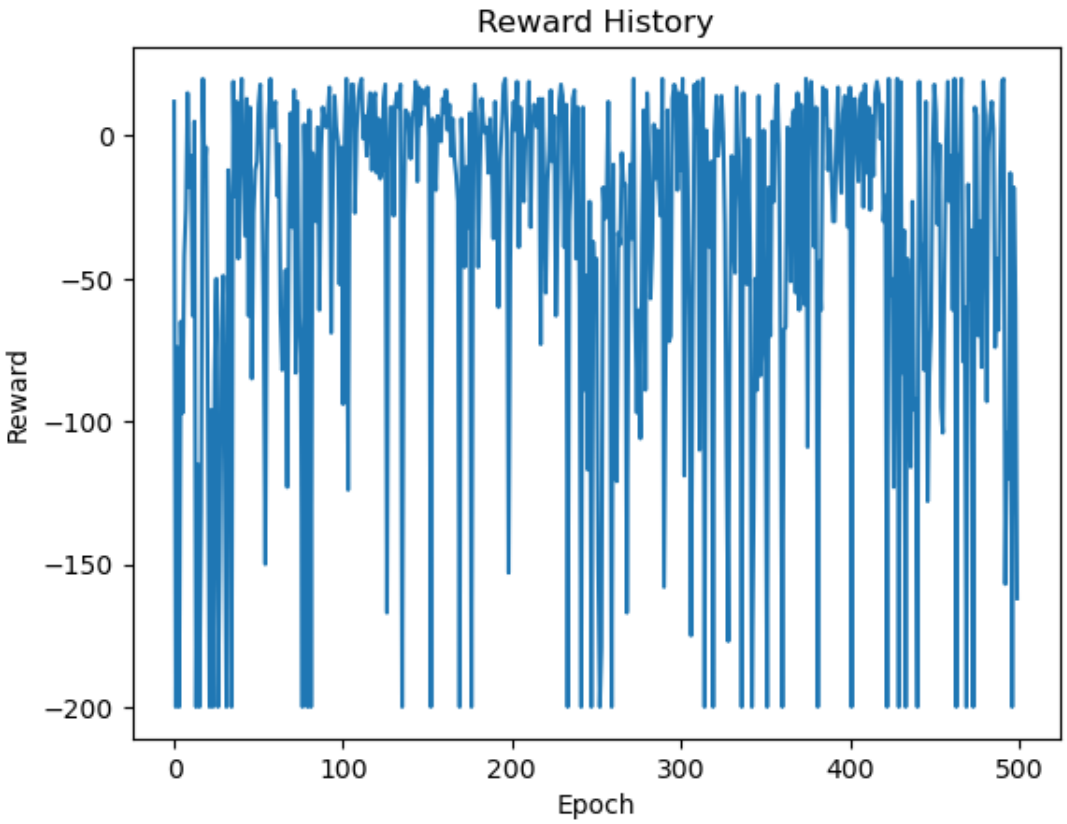


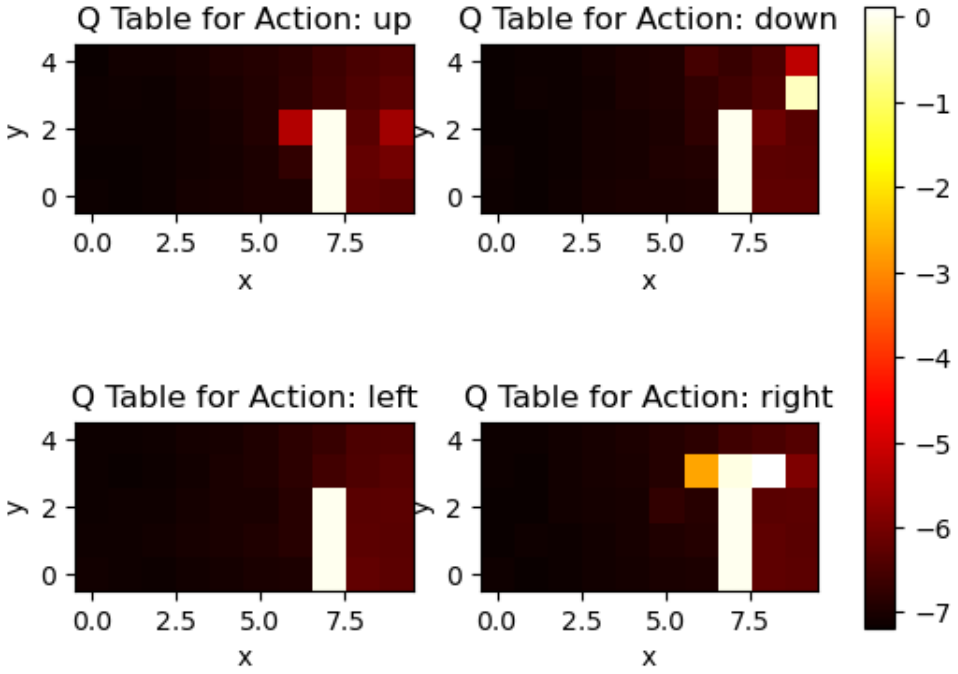


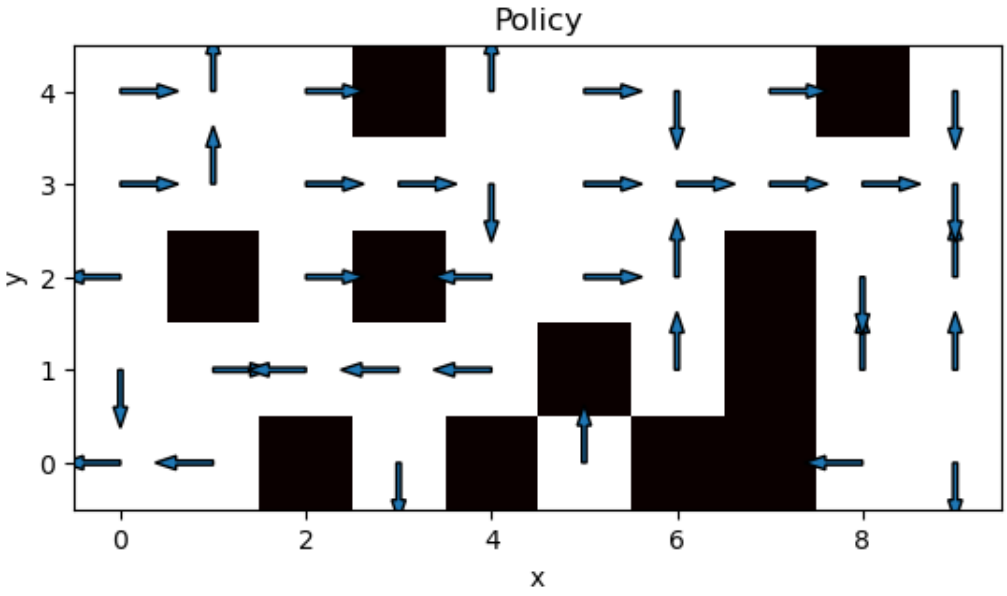
[23]:
```
## Experiments for Q-learning with random door

rng_door = True # random door
```

```
# Initialize learner
qAgent_rng_door = qLearner(epochs, time_steps, alpha, gamma, epsilon,␣
 ↪epsilon_decay, rng_door)

# Run experiment
qAgent_rng_door.run(verbose=True)

# Generate plots
qAgent_rng_door.plot_reward_history()
qAgent_rng_door.plot_Q_table()
qAgent_rng_door.plot_policy()

# Show plots
plt.show()
```

```
Epoch 1/500 | Reward for epoch: 12
Epoch 101/500 | Reward for epoch: -94
Epoch 201/500 | Reward for epoch: 1
Epoch 301/500 | Reward for epoch: -12
Epoch 401/500 | Reward for epoch: 17
Epoch 500/500 | Reward for epoch: -162
```




Q Table for Action: up
Q Table for Action: down
Q Table for Action: left
Q Table for Action: right
y
x
0.0
2.5
5.0
7.5
0
2
4
0
−1
−2
−3
−4
−5
−6
−7


Policy
y
x
0
1
2
3
4
0
2
4
6
8

[24]:
```
# Overlay learning curves
plt.figure("Learning curves")
plt.plot(sarsaAgent_rng_door.reward_history, label="SARSA")
plt.plot(qAgent_rng_door.reward_history, label="Q-learning")
plt.xlabel("Epochs")
plt.ylabel("Reward")
plt.title("Learning curves Q-learning vs SARSA with random door")
plt.legend()
plt.show()
```