# CS 352 Reinforcement Learning Project II

Saad Abdul Hakim Qureshi
CS 2024

**Abstract— in this paper, we will apply the Sarsa and Q Learning algorithms on the Taxi Environment and compare both of them in terms of efficiency. Furthermore we also study which algorithm results in the more average reward.**

## 1. Introduction

In Reinforcement Learning, the two most popular algorithms for solving Markov Decision Processes (MDPs) are Sarsa and Q-learning. Sarsa and Q-learning are both tabular, model-free, and temporal difference methods used to learn the optimal policy in an MDP. Both methods are based on the Bellman equation, but they differ in the way they update their action-value estimates. SARSA, the Q-value of the next state-action pair is used to update the Q-value of the current state-action pair, while in Q-learning, the maximum Q-value of the next state is used to update the Q-value of the current state-action pair.

There are four designated locations in the grid world indicated by Red), Green), Yellow), and Blue). When the episode starts, the taxi starts off at a random square and the passenger is at a random location. The taxi drives to the passenger's location, picks up the passenger, drives to the passenger's destination (another one of the four specified locations), and then drops off the passenger. Once the passenger is dropped off, the episode ends. The number of states are 600 and there 6 possible actions for each state.

The objective of this paper is to compare the computational efficiency and average reward maximization of Sarsa and Q-learning using the Taxi Environment from OpenAI as our Grid World.

## 2. Experimental Setup

Both of the algorithms are implemented using python in jupyter notebook on VS Code. The Grid World is taken from the Taxi Environment provided by OpenAI.

The libraries used are:

1. Numpy
2. Mathplotlib
3. Random
4. Gym(Taxi V3)
5. Tabulate

```
import random
import numpy as np
import time
import matplotlib.pyplot as plt
import gym

# Connecting to OpenAI
env = gym.make('Taxi-v3')
```

Fig 1. Setup

## 3. Task 1 Implementation

Both algorithms functions take learning rate and gamma as input and return the total number of episodes each algorithm runs before converging on the policy for the Taxi Environment.

```python
def sarsa(alpha, gamma):
    # Initializing Q arbitarrialy.
    for i in range(env.observation_space.n):
        for j in range(env.action_space.n):
            Q[i, j] = round(random.random(),1)

    episodes = 0

    # Running algorithm till convergence.
    while True:
        count = 0

        # Initializing s.

        s = env.reset()[0]

        # Taking actiona, observing r and s'.
        a = np.argmax(Q[s])
        tuple = env.step(a)
        next_s = tuple[0]
        R = tuple[1]
        done = tuple[2]
        # Saving old policiy to check for convergence.
        old_Q = np.zeros_like(Q)
        for i in range(env.observation_space.n):
            for j in range(env.action_space.n):
                old_Q[i, j] = Q[i,j]

        # Repeating episodes until s is terminal or MAX_STEPS are reached.
        while not done and count < 20:
            # print(Q)

            next_a = np.argmax(Q[next_s])# Greedy choice of next action.

            #Updating Q. Removing assignment part since that can cause values of the policy to become significantly large.
            Q[s, a] += alpha * (R + gamma * Q[next_s, next_a] - Q[s, a])

            # Setting next state and action to current.
            s = next_s
            a = next_a

            # Taking action a, observing r and s'
            tuple = env.step(a)
            next_s = tuple[0]
            R = tuple[1]
            done = tuple[2]
            count += 1

        episodes += 1

        convergence = True
        for s in range(env.observation_space.n):
            if np.sum(np.abs(Q[s]-old_Q[s])) > 1e-2:
                convergence = False

        if convergence:
            return episodes
```

Fig2. Sarsa Algorithm

```python
def q_learning(alpha, gamma):
    # Initializing Q arbitarrialy.
    for i in range(env.observation_space.n):
        for j in range(env.action_space.n):
            Q[i, j] = round(random.random(),1)

    episodes = 0
    # Running algorithm till convergence.
    while True:
        count = 0
        # Initializing s.
        s = env.reset()[0]

        # Taking actiona, observing r and s'.
        a = np.argmax(Q[s])
        tuple = env.step(a)
        next_s = tuple[0]
        R = tuple[1]
        done = tuple[2]
        # Saving old policiy to check for convergence.
        old_Q = np.zeros_like(Q)
        for i in range(env.observation_space.n):
            for j in range(env.action_space.n):
                old_Q[i, j] = Q[i,j]

        # Repeating episodes until s is terminal or MAX_STEPS are reached.
        while not done and count < 20:

            #Updating Q. Removing assignment part since that can cause values of the policy to become significantly large.
            Q[s, a] += alpha * (R + gamma * np.max(Q[next_s]) - Q[s, a])
            # Setting next state to current
            s = next_s

            # Taking actiona, observing r and s'.
            a = np.argmax(Q[s])
            tuple = env.step(a)
            next_s = tuple[0]
            R = tuple[1]
            done = tuple[2]
            count += 1

        episodes += 1

        convergence = True
        for s in range(env.observation_space.n):
            if np.sum(np.abs(Q[s]-old_Q[s])) > 1e-2:
                convergence = False

        if convergence:
            return episodes
```

Fig3. Q-Learning Algorithm

To check for convergence the algorithm compares the old policy with the new one and checks if the difference is negligible or not.

```python
convergence = True
for s in range(env.observation_space.n):
    if np.sum(np.abs(Q[s]-old_Q[s])) > 1e-2:
        convergence = False

if convergence:
    return episodes
```

Fig4. Convergence Check

Furthermore since the episodes generated are arbitrary there is a high chance of the agent not reaching the terminal states which is why we have modified the approach where each episode has a maximum time step of 20.

```python
# Repeating episodes until s is terminal or MAX_STEPS are reached.
while not done and count < 20:

    #Updating Q. Removing assignment part since that can cause valu
    Q[s, a] += alpha * (R + gamma * np.max(Q[next_s]) - Q[s, a])
    # Setting next state to current
    s = next_s

    # Taking actiona, observing r and s'.
    a = np.argmax(Q[s])
    tuple = env.step(a)
    next_s = tuple[0]
    R = tuple[1]
    done = tuple[2]
    count += 1

episodes += 1
```

Fig5. Max Steps

We use the following calls provided by OpenAI gym to get our starting_state, NUM_STATES, NUM_ACTION, next_state, next_action variables.

```python
# Initializing Q arbitarrialy.
for i in range(env.observation_space.n):
    for j in range(env.action_space.n):
        Q[i, j] = round(random.random(),1)

episodes = 0
# Running algorithm till convergence.
while True:
    count = 0
    # Initializing s.
    s = env.reset()[0]

    # Taking actiona, observing r and s'.
    a = np.argmax(Q[s])
    tuple = env.step(a)
    next_s = tuple[0]
    R = tuple[1]
    done = tuple[2]
```

Fig6. OpenAI Calls

## 4. Task 1 Computational Efficiency Results

The following diagrams shows the total number of episodes (iterations) generated and time taken by both algorithms before converging with respect to the learning rate (alpha). The discount factor gamma has been kept constant to 0.5.

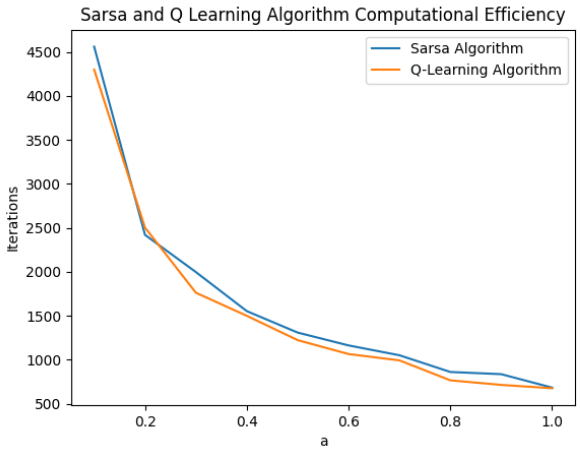


Fig7. Computational Efficiency (Episodes)

Parameters:

0.1 < Alpha < 1.0 (++0.1)
800 < Episodes (Iterations) < 4500
Gamma = 0.5

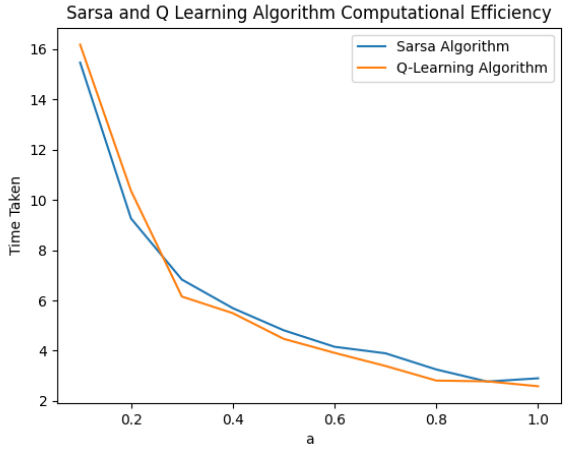


Fig8. Computational Efficiency (Time Taken)

Parameters:

0.1 < Alpha < 1.0 (++0.1)
0 < Time Taken < 16
Gamma = 0.5

From the diagrams, we can see that both Q-learning and Sarsa are almost computationally the same in terms of efficiency which can be as a result of both algorithms having similar action value estimates. Sarsa initially performs slightly better for smaller learning rates then Q Learning takes the lead for larger learning rates.

Furthermore, we also notice that the learning rate affects the stability of both algorithm. A high learning rate can cause the algorithm to converge faster but can also causes greater variation and oscillations in the number of episodes being generated. A low learning rate can improve stability but leads to slow convergence indicated by the greater number of episodes.

## 5. Task 2 Implementation

Following are the values for learning rate, epsilon and exploration rate.

alpha = 0.9(Faster convergence)
gamma = 0.99(Irrelevant)
epsilon = 0.1(Giving more control to Greedy Policy)
episodes = 1000(Same number of rewards sums collected for each episode in both algorithms)

The algorithms have been modified to check for epsilon instead of max steps to stop the episode in case for terminal state not being reached.

Instead of returning the number of episodes both function now a list of the sum of rewards for each episode from which the average is taken. The result is displayed in a graph and table at the end of the code.

```
def sarsa(alpha, gamma, epsilon):
    rewards = []
    # Initializing Q arbitarrialy.
    for i in range(env.observation_space.n):
        for j in range(env.action_space.n):
            Q[i, j] = round(random.random(),1)


    # Running algorithm till convergence.
    for i in range(1000):
        sum = 0
        # Initializing s.

        s = env.reset()[0]

        # Taking actiona, observing r and s'.
        a = epsilon_greedy(s, epsilon)
        tuple = env.step(a)
        next_s = tuple[0]
        R = tuple[1]
        done = tuple[2]
        # Repeating episodes until s is terminal
        while not done:
            sum+=R

            next_a = epsilon_greedy(next_s, epsilon)# Greedy choice of next action.

            #Updating Q. Removing assignment part since that can cause values of the policy to become significantly large.
            Q[s, a] += Q[s, a] + alpha * (R + gamma * Q[next_s, next_a] - Q[s, a])

            # Setting next state and action to current.
            s = next_s
            a = next_a

            # Taking action a, observing r and s'
            tuple = env.step(a)
            next_s = tuple[0]
            R = tuple[1]
            done = tuple[2]
        rewards.append(sum)
    return rewards
```

Fig9. Sarsa Algorithm

```
def q_learning(alpha, gamma, epsilon):
    rewards = []
    # Initializing Q arbitarrialy.
    for i in range(env.observation_space.n):
        for j in range(env.action_space.n):
            Q[i, j] = round(random.random(),1)

    # Running algorithm till convergence.
    for i in range(1000):
        sum = 0
        # Initializing s.
        s = env.reset()[0]

        # Taking actiona, observing r and s'.
        a = epsilon_greedy(s, epsilon)
        tuple = env.step(a)
        next_s = tuple[0]
        R = tuple[1]
        done = tuple[2]
        # Repeating episodes until s is terminal
        while not done:
            sum+=R
            #Updating Q. Removing assignment part since that can cause values of the policy to become significantly large.
            Q[s, a] += Q[s, a] + alpha * (R + gamma * np.max(Q[next_s]) - Q[s, a])
            # Setting next state to current
            s = next_s

            # Taking actiona, observing r and s'.
            a = epsilon_greedy(s, epsilon)
            tuple = env.step(a)
            next_s = tuple[0]
            R = tuple[1]
            done = tuple[2]

        rewards.append(sum)

    return rewards
```

Fig10. Q-Learning Algorithm

To prevent the episodes from running forever we are now modifying the greedy policy to randomly select an action based on epsilon value. We have kept the epsilon value low to keep the control of the algorithm in the greedy policy.

```
# Epsilon-greedy policy
def epsilon_greedy(state, epsilon):
    if np.random.uniform(0, 1) < epsilon:
        action = env.action_space.sample()
    else:
        action = np.argmax(Q[state])
    return action
```

Fig11. Q-Learning Algorithm

## 6. Task 2 Average Rewards Result

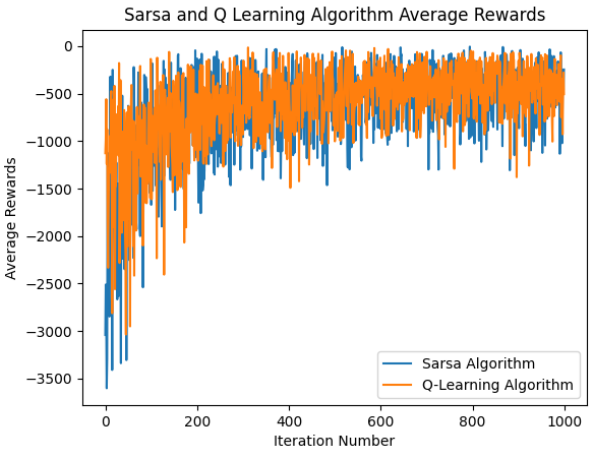


Fig12. Average Rewards Graph

| Algorithm | Average Reward |
|---|---|
| SARSA | -667.373 |
| Q-learning | -608.375 |

Fig13. Average Rewards

From the graph we can see that the Sarsa algorithm results in the more average case initially but as the policy converges the difference between these Sarsa and Q Learning becomes insignificant as both are closer to their average rewards.

## 7. Discussion and Conclusion

From the results obtained in this study, we can conclude that both Sarsa and Q-learning are effective in solving the Taxi Environment problem. However, the choice of algorithm will depend on the specific requirements of the problem. In terms of computational efficiency, both algorithms perform almost equally, with slight variations depending on the learning rate.

In terms of average rewards, our results showed that Sarsa initially results in a higher average reward, but as the policy converges, the difference between Sarsa and Q-learning becomes insignificant, with both algorithms having similar average rewards.

It is important to note that the choice of hyperparameters, such as learning rate and epsilon, can greatly impact the performance of the algorithms. A higher learning rate can cause the algorithm to converge faster but may lead to instability and oscillations. A lower learning rate can improve stability but may result in slower convergence. Similarly, a higher epsilon value can result in greater exploration, but may lead to slower convergence, while a lower epsilon value may result in a more exploitative policy, but may miss out on potential rewards.

## 8. Appendix:

- Taxi Environment: https://www.gymlibrary.dev/environments/toy_text/taxi/
- GitHub: https://github.com/saadabdulhakimqureshi?tab=repositories

## 9. References:

**None.**