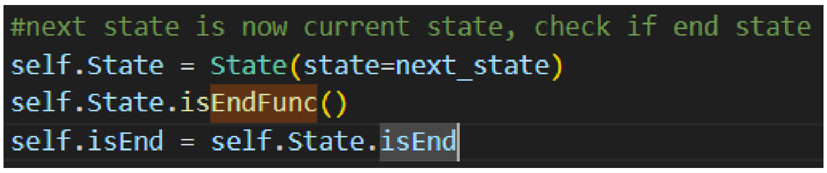
Computational Bottlenecks in FrozenLake Q-Learning Code

I have used ChatGPT 4o model LLM to identify the bottlenecks and these are the following bottlenecks it identified:

1. Redundant State Object Recreation

- Problem: New State object is created every step.



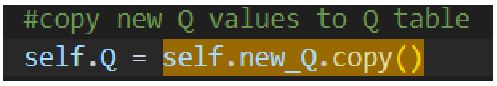
```
#next state is now current state, check if end state
self.State = State(state=next_state)
self.State.isEndFunc()
self.isEnd = self.State.isEnd
```

- Impact: Medium slowdown for 10,000+ episodes.

- Optimization: Reuse the existing State object by updating its attributes.

2. Excessive Copying of Q-tables

- Problem: Full Q-table copied every step.



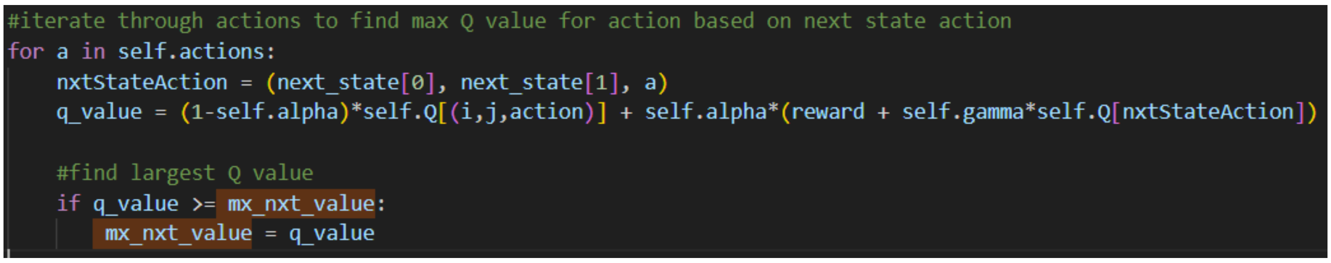
```
#copy new Q values to Q table
self.Q = self.new_Q.copy()
```

- Impact: Mild now, but serious for larger boards.

- Optimization: Update Q-values directly without full copy.

3. Inefficient Finding of Max Q-value

- Problem: Manual linear scan through actions.



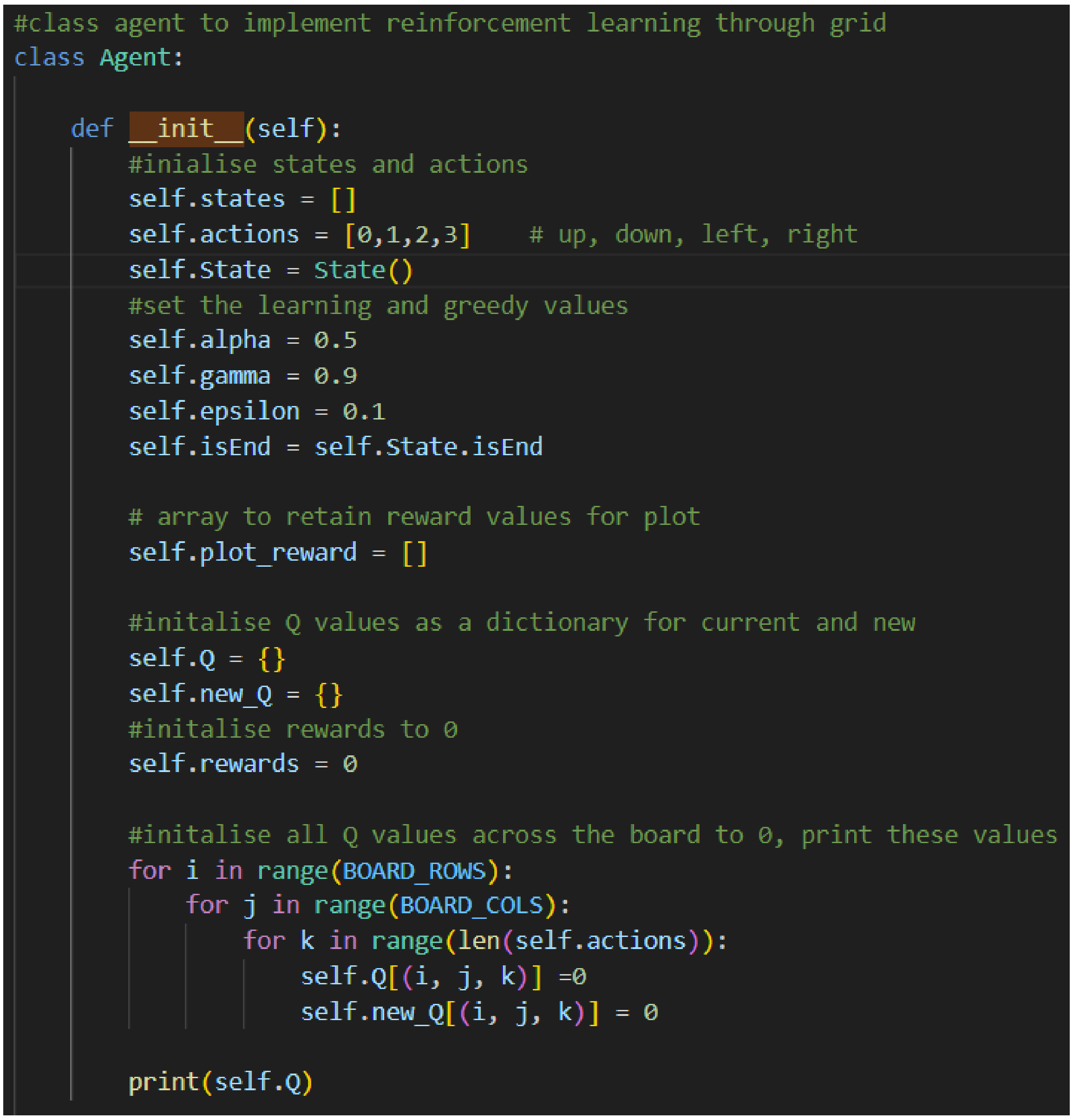
```
#iterate through actions to find max Q value for action based on next state action
for a in self.actions:
    nxtStateAction = (next_state[0], next_state[1], a)
    q_value = (1-self.alpha)*self.Q[(i,j,action)] + self.alpha*(reward + self.gamma*self.Q[nxtStateAction])

    #find largest Q value
    if q_value >= mx_nxt_value:
        mx_nxt_value = q_value
```

- Impact: Fine now, serious if actions >10.

- Optimization: Use numpy arrays and np.argmax().

4. Unnecessary Printing at Startup

- Problem: Printing full Q-table at initialization.



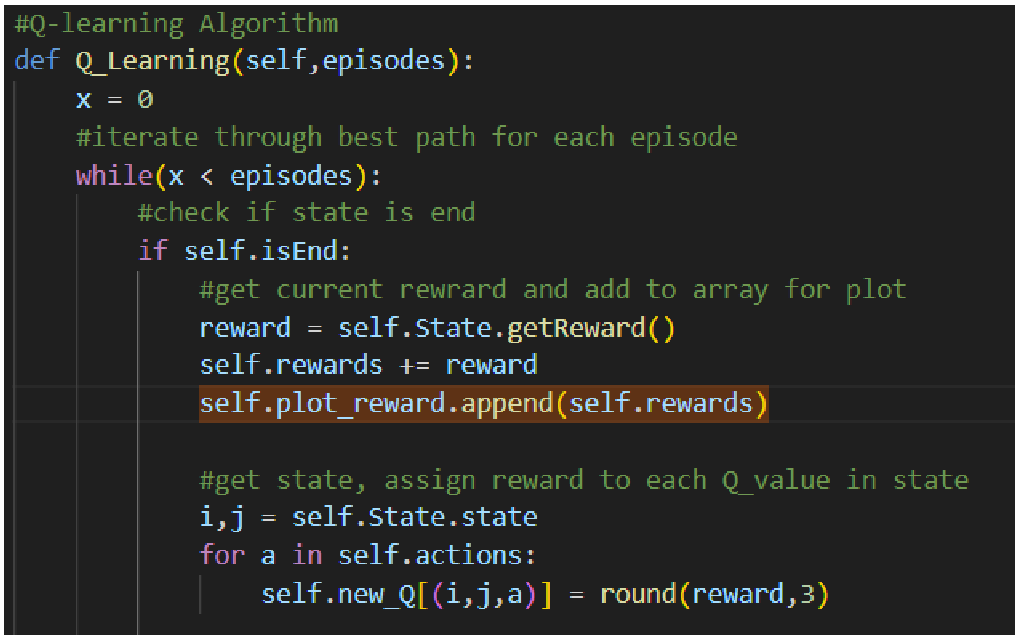
```python
#class agent to implement reinforcement learning through grid
class Agent:

    def __init__(self):
        #inialise states and actions
        self.states = []
        self.actions = [0,1,2,3]    # up, down, left, right
        self.State = State()
        #set the learning and greedy values
        self.alpha = 0.5
        self.gamma = 0.9
        self.epsilon = 0.1
        self.isEnd = self.State.isEnd

        # array to retain reward values for plot
        self.plot_reward = []

        #initalise Q values as a dictionary for current and new
        self.Q = {}
        self.new_Q = {}
        #initalise rewards to 0
        self.rewards = 0

        #initalise all Q values across the board to 0, print these values
        for i in range(BOARD_ROWS):
            for j in range(BOARD_COLS):
                for k in range(len(self.actions)):
                    self.Q[(i, j, k)] =0
                    self.new_Q[(i, j, k)] = 0

        print(self.Q)
```

- Impact: Slows startup slightly.

- Optimization: Remove or comment out printing.

5. Reward Plotting Inefficiency

- Problem: Appending every episode's reward individually.



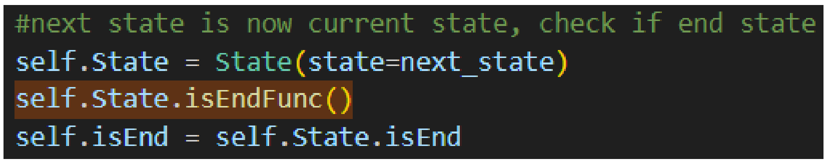
```
#Q-learning Algorithm
def Q_Learning(self,episodes):
    x = 0
    #iterate through best path for each episode
    while(x < episodes):
        #check if state is end
        if self.isEnd:
            #get current rewrard and add to array for plot
            reward = self.State.getReward()
            self.rewards += reward
            self.plot_reward.append(self.rewards)

            #get state, assign reward to each Q_value in state
            i,j = self.State.state
            for a in self.actions:
                self.new_Q[(i,j,a)] = round(reward,3)
```

- Impact: Small, but noticeable with huge episode counts.

- Optimization: Batch rewards or smooth plots.

6. Redundant isEnd Checking

- Problem: Manual function call to check end state.



```
#next state is now current state, check if end state
self.State = State(state=next_state)
self.State.isEndFunc()
self.isEnd = self.State.isEnd
```

- Impact: Small overhead.

- Optimization: Automate end checking inside methods.