CSE 584-Homework2

Name: Samira Malek PSU ID: sxm6547

The selected is as following:

```python
import numpy as np
import gym

env = gym.make("FrozenLake-v1", is_slippery=True)

alpha = 0.8           # Learning rate
gamma = 0.95          # Discount factor
epsilon = 1.0         # Exploration rate
epsilon_decay = 0.995 # Epsilon decay factor
min_epsilon = 0.01    # Minimum exploration probability
episodes = 10000      # Total episodes
max_steps = 100       # Max steps per episode

q_table = np.zeros([env.observation_space.n, env.action_space.n])

for episode in range(episodes):
    state = env.reset()
    done = False
    steps = 0

    for step in range(max_steps):
        if np.random.rand() < epsilon:
            action = env.action_space.sample()
        else:
            action = np.argmax(q_table[state])

        next_state, reward, done, _ = env.step(action)

        old_value = q_table[state, action]
        next_max = np.max(q_table[next_state])
        q_table[state, action] = old_value + alpha * (reward + gamma * next_max - old_value)

        state = next_state
        steps += 1

        if done:
            break
    epsilon = max(min_epsilon, epsilon * epsilon_decay)

total_episodes = 100
total_success = 0

for episode in range(total_episodes):
    state = env.reset()
    done = False
    steps = 0

    while not done:
        action = np.argmax(q_table[state])
        state, reward, done, _ = env.step(action)
        steps += 1

        if done and reward == 1:
            total_success += 1

print(f"Success rate over {total_episodes} episodes: {total_success / total_episodes * 100}%")

```

1-Writing an abstract that provides a high-level overview of the code's purpose and the overall process it follows

This code implements a reinforcement learning (RL) algorithm using Q-learning to train an agent in the Frozen Lake environment, a grid-based problem where the agent aims to navigate to a goal while avoiding pitfalls. The algorithm uses a Q-table, which stores state-action values, to learn the best action for each state. The process begins by initializing parameters such as the learning rate, discount factor, and exploration rate. Through a series of episodes, the agent interacts with the environment, selecting actions using an epsilon-greedy approach that balances exploration and exploitation.

At each step, the Q-table is updated based on the Q-learning formula, incorporating the immediate reward and the estimated value of future states. This learning process is repeated over numerous episodes, with the exploration rate gradually decreasing to favor exploiting learned strategies. Once trained, the agent’s performance is evaluated by running test episodes to determine its success rate in reaching the goal. This approach provides a foundational method for solving RL problems with discrete states and actions, demonstrating the effectiveness of Q-learning in learning optimal policies through trial and error.

2-Identifying the core section (such as a couple of functions about the RL implementations) of the reinforcement learning implementation and add comments to each line, explaining what it is doing. This demonstrates your step-by-step understanding.

```
# Loop over each episode for training
for episode in range(episodes):
    state = env.reset()         # Reset environment and get initial state
    done = False                # Flag to track if the episode is done
    steps = 0                   # Counter to track the number of steps in the episode

    # Loop for each step within an episode (up to max_steps)
    for step in range(max_steps):
        # Select action using epsilon-greedy strategy
        if np.random.rand() < epsilon:       # With probability epsilon, explore
            action = env.action_space.sample()  # Take a random action (explore)
        else:
            action = np.argmax(q_table[state])  # Otherwise, choose the best-known action (exploit)

        # Perform the action and observe the outcome (next state, reward, and done flag)
        next_state, reward, done, _ = env.step(action)

        # Get the Q-value for the current state-action pair (Q(s, a))
        old_value = q_table[state, action]

        # Find the maximum Q-value for the next state (best action in the next state)
        next_max = np.max(q_table[next_state])

        # Update the Q-value for the current state-action pair using the Q-learning formula
        # Q(s, a) ← Q(s, a) + α [reward + γ * max_a Q(s', a) - Q(s, a)]
        q_table[state, action] = old_value + alpha * (reward + gamma * next_max - old_value)

        # Move to the next state
        state = next_state
        steps += 1

        # If we reach a terminal state (goal or fall in a hole), end the episode
        if done:
            break

    # Decay epsilon after each episode to reduce exploration over time
    epsilon = max(min_epsilon, epsilon * epsilon_decay)  # Ensures epsilon does not fall below min_epsilon
```