Assignment 2

In the last assignment, we implemented a deterministic version of the *Frozen Lake* environment. In this assignment, you will add stochastic dynamics, and solve it with value iteration, linear programming, and Q-learning.

Slippery Frozen Lake

The environment is the same as in assignment 1, but the actions will result in a stochastic outcome. You are given a 4×4 grid with each cell labeled as S (start), F (frozen), H (hole), or G (goal) in the following configuration:

```
"4x4":[
    "SFFF",
    "FHFH",
    "FFFH",
    "HFFG"
]
```

Each cell in the grid can be considered as a "state", and the states can be numbered as follows:

```
0  1  2  3
4  5  6  7
8  9 10 11
12 13 14 15
```

For each state, there are four possible actions: N (north), E (east), W (west), and S (south). For every state except those labeled with H and G, when an action is taken, the agent moves in the corresponding direction with probability $1/3$, and in the two perpendicular directions with probability $1/3$ each. If the agent encounters a wall, the action will keep the agent in the current state. For example, starting from state 10, choosing action N will result in the agent moving to state 6 (north), state 9 (west), state 11 (east) with equal probability. The states labeled H and G are sink states, i.e. every action keeps the agent in the same state.

A reward of 1 should be given for the first time reaching the goal state (every transition from a non-goal state to the goal state is given a reward of 1), a reward of 0 for all other transitions.

Part I

Problems:

1. Use value iteration to compute the optimal values and policy with discount factor $\lambda = 0.99$. Use $\varepsilon = 10^{-3}$.
2. Use linear programming to compute the optimal values and policy with discount factor $\lambda = 0.99$.

In [117]:
```
import numpy as np
from pulp import *

# Define the environment
grid_size = 4
states = np.arange(grid_size ** 2)

print(f"states: {states}")

actions = {
    'N': -grid_size,
    'E': 1,
    'W': -1,
    'S': grid_size}

print(f"actions: {actions}")

action_nums = ['N', 'E', 'W', 'S'] #for easy retrieval

# Rewards and discount factor
R = np.zeros(len(states))
R[-1] = 1  # Goal state reward

print(f"Reward: {R}\n")

lambda_ = 0.99
epsilon = 1e-3

holes = [5, 7, 11, 12]

# Initialize value function to zeros and then use the Banach's Theorem to
V = np.zeros(len(states))

# Transition function
# input: (state-action pair)
# output: next_state
def transition(state, action):
    if (state == 15) or (state in holes):  # Goal or holes state
        return state
    new_state = state + actions[action]
    if new_state < 0 or new_state >= len(states) or (state % grid_size ==
        return state  # Return to same state if out of bounds or invalid
    return new_state

def perpendicular(action):
    if action in ['N', 'S']:
        return ['E', 'W']
    elif action in ['E', 'W']:
        return ['N', 'S']

# Value Iteration
# input: V (initially zeros), {R, states, actions, lambda_, epsilon} (as
# output: Optimal value function (state -> Real number) {From the map ch
def value_iteration(V, R, states, actions, lambda_, epsilon):
#     print(f"value:{V}. \nReward: {R}.\n")
    delta = float('inf')
    while delta > epsilon:
```

```
        delta = 0
        for s in states:
            if s == 15 or s in holes:  # Goal or holes state
                continue
            v = V[s]
            action_probability_value = np.zeros(len(actions))
            for j, a in enumerate(actions):
                next_state = transition(s, a)
                value_main_action = (1/3) * (R[next_state] + lambda_ * V
                value_perpendicular_action = np.zeros(2)
                for i,p in enumerate(perpendicular(a)):
                    next_state = transition(s, p)
                    value_perpendicular_action[i] = (1/3) * (R[next_stat
                action_probability_value[j] = sum([value_main_action, va
            V[s] = max(action_probability_value)
            delta = max(delta, abs(v - V[s]))
    return V

# Run value iteration
V_optimal = value_iteration(V, R, states, actions, lambda_, epsilon)

print('Optimal Value Function:')
print(V_optimal)


# Linear Programming formulation
def linear_programming():
    num_states = len(states)
    num_actions = len(actions)
    lp_prob = LpProblem("FrozenLakeLP", LpMinimize)
    V = LpVariable.dicts("V", range(num_states), lowBound=0)

    # Objective Function: Minimize the sum of the state values: (V0 + V1
    lp_prob += lpSum([V[s] for s in range(num_states)])

    # Constraints based on the Bellman equation
    for s in range(num_states):
        action_probability_value = [0]*len(actions)
        for j, a in enumerate(actions):
            next_state = transition(s, a)
            value_main_action = (1/3) * (R[next_state] + lambda_ * V[nex
            value_perpendicular_action = [0]*2
            for i,p in enumerate(perpendicular(a)):
                next_state = transition(s, p)
                value_perpendicular_action += (1/3) * (R[next_state] + la
            action_probability_value = value_main_action + value_perpend
            lp_prob += V[s] >= action_probability_value #important const

    # Solve the LP problem
    lp_prob.solve(PULP_CBC_CMD(msg=0))
    v_values = np.array([value(V[i]) for i in range(num_states)])
    return v_values


# Linear Programming
print("\nLinear Programming optimal value function:")
v_values = linear_programming()
```

```
print(v_values)


# Extracting the policy from the optimal value function
# input: V (optimal), states, actions, lambda_
# output: optimal policy (states -> actions) {Easily find the action to 
def extract_policy(V, states, actions, lambda_):
    policy = {}
    for s in states:
        if s == 15:  # Goal state
            policy[s] = 'G'
            continue
        if s in holes:
            policy[s] = 'H'
            continue
        action_probability_value = np.zeros(len(actions))
        for j, a in enumerate(actions):
            next_state = transition(s, a)
            value_main_action = (1/3) * (R[next_state] + lambda_ * V[nex
            value_perpendicular_action = np.zeros(2)
            for i,p in enumerate(perpendicular(a)):
                next_state = transition(s, p)
                value_perpendicular_action[i] = (1/3) * (R[next_state] +
            action_probability_value[j] = sum([value_main_action, value_
        best_action = action_nums[np.argmax(action_probability_value)]
        policy[s] = best_action
    return policy

# Extract policy
policy_optimal = extract_policy(V_optimal, states, actions, lambda_)

print('Optimal Policy:')
print(policy_optimal)


policy_optimal_lp = extract_policy(v_values, states, actions, lambda_)
print("\nLinear Programming optimal policy:")
print(policy_optimal_lp)
```

```
states: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
actions: {'N': -4, 'E': 1, 'W': -1, 'S': 4}
Reward: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]

Optimal Value Function:
[0.53006594 0.48303111 0.45230256 0.43713666 0.54783225 0.
 0.35040218 0.         0.58314889 0.63686767 0.6097876  0.
 0.         0.73744109 0.86067745 0.        ]

Linear Programming optimal value function:
[ 54.202593  49.880319  47.069569  45.68517   55.845096   0.
  35.834807   0.        59.179874  64.307982  61.520756   0.
   0.        74.172044  86.283743 100.      ]
Optimal Policy:
{0: 'W', 1: 'N', 2: 'N', 3: 'N', 4: 'W', 5: 'H', 6: 'E', 7: 'H', 8:
'N', 9: 'S', 10: 'W', 11: 'H', 12: 'H', 13: 'E', 14: 'S', 15: 'G'}

Linear Programming optimal policy:
{0: 'W', 1: 'N', 2: 'N', 3: 'N', 4: 'W', 5: 'H', 6: 'E', 7: 'H', 8:
'N', 9: 'S', 10: 'W', 11: 'H', 12: 'H', 13: 'E', 14: 'S', 15: 'G'}
```

Part II

In this part, you will use Q-learning to compute the optimal values and policy. We will first need to create a verison of the environment that can be sampled from.

The Frozen Lake Environment

We will convert *Frozen Lake* into an environment in the style of [Open-AI Gym (https://www.gymlibrary.dev/content/basic_usage/)](https://www.gymlibrary.dev/content/basic_usage/) (we will not need all of the parts of OpenAI Gym). This will mean the environment will have a `reset` and `step` functions:

- `reset()` takes no inputs and will return the initial state.
- `step(action)` will take an action as input, sample a transition in the environment, and will return five values: the next state, the reward, the truncated signal, the terminated signal, and an information dictionary for debugging.

The terminated signal indicates if the episode has reached the special termination state, a sink state of zero value (see [Section 3.3 of Sutton (http://incompleteideas.net/book/RLbook2020.pdf#%5B%7B%22num%22%3A875%2C%22gen](http://incompleteideas.net/book/RLbook2020.pdf#%5B%7B%22num%22%3A875%2C%22gen) `Terminated` should be true when either a hole or the goal is reached, and false otherwise. The truncated signal indicated that the episode should be reset due to some external reason, like a timeout. `Truncated` should be true when 100 steps have elapsed.

When either `terminated` or `truncated` is true, the agent should go to the next episode by resetting. The information dictionary can be empty (or contain whatever you want), as the agent is not allowed to use it. The current state of the environment will be remembered inside of the object, i.e. the environment will only receive the action sequence.

When `terminated` is true, people typically explicitly set the next state value to zero, i.e. by using `(1-terminated)*discount*next_state_value` instead of simply `discount*next_state_value`. You can decide whether or not you want to do this in your code.

Implement the *Frozen Lake* environment with the same dynamics as in Part I. The states and actions should be integers (so that we can use it to index into the Q-table later). Also add `num_states` and `num_actions` to the class, which contain the number of states and the number of actions. You may find [numpy.random.choice (https://numpy.org/doc/stable/reference/random/generated/numpy.random.choice.html)](https://numpy.org/doc/stable/reference/random/generated/numpy.random.choice.html) useful for sampling, but you can use whatever you want.

Problem:

1. Implement the *Frozen Lake* environment described above. The test code below should be able to run without errors.

In [158]:
```
### Test the environment ###
import numpy as np

def modified_transition(state, action):
    if state == 15:  # Goal state
        return state
    if state in holes:
        return state
    sample_vec = perpendicular(action) + [action]
    sampled_action = np.random.choice(sample_vec, 1, p=[1/3, 1/3, 1/3])[(
    new_state = state + actions[sampled_action]
    if new_state < 0 or new_state >= len(states) or (state % grid_size ==
        return state  # Return to same state if out of bounds or invalid
    return new_state

class FrozenLakeEnvironment:
    def __init__(self, grid_size = 4, lambda_ = 0.99, epsilon = 1e-3, ho
        self.grid_size = 4
        self.states = np.arange(grid_size ** 2)
        self.num_states = len(self.states)

        self.cur_state = 0
        self.timestep = 0
        self.info = {}
        self.terminated = False
        self.truncated = False
        self.action_nums = ['N', 'E', 'W', 'S']
        self.actions = {
            'N': -grid_size,
            'E': 1,
            'W': -1,
            'S': grid_size}

        self.num_actions = len(self.actions)

#         print(f"actions: {actions}")

        # Rewards and discount factor
        self.R = np.zeros(len(self.states))
        self.R[-1] = 1  # Goal state reward

        print(f"Reward: {self.R}\n")

        self.lambda_ = lambda_
        self.epsilon = epsilon

        self.holes = [5, 7, 11, 12]


    def reset(self):
        self.cur_state = self.states[0]
        self.timestep = 0
        return self.cur_state

    def step(self, action):
        self.timestep += 1
        self.cur_state = modified_transition(self.cur_state, self.action_
```

```
        if self.cur_state in holes or self.cur_state == 15:
            self.terminated = True
        if self.timestep >= 1000:
            self.truncated = True
        reward = self.R[self.cur_state]
        return [self.cur_state, reward, self.terminated, self.truncated,


env = FrozenLakeEnvironment()

print("Number of states:", env.num_states)
print("Number of actions:", env.num_actions)
print("Initial state:", env.reset())
for n in range(10):
    rand_action = np.random.randint(env.num_actions)
    S, r, terminated, truncated, info = env.step(rand_action)
    print("  Step", n)
    print("    S =", S)
    print("    A =", rand_action)
    print("    R =", r)
    print("    terminated =", terminated)
    print("    truncated =", truncated)

    if terminated or truncated:
        break
```

```
Reward: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]

Number of states: 16
Number of actions: 4
Initial state: 0
  Step 0
    S = 0
    A = 2
    R = 0.0
    terminated = False
    truncated = False
  Step 1
    S = 0
    A = 0
    R = 0.0
    terminated = False
    truncated = False
  Step 2
    S = 1
    A = 3
    R = 0.0
    terminated = False
    truncated = False
  Step 3
    S = 2
    A = 3
    R = 0.0
    terminated = False
    truncated = False
  Step 4
    S = 1
    A = 0
    R = 0.0
    terminated = False
    truncated = False
  Step 5
    S = 0
    A = 2
    R = 0.0
    terminated = False
    truncated = False
  Step 6
    S = 0
    A = 0
    R = 0.0
    terminated = False
    truncated = False
  Step 7
    S = 4
    A = 3
    R = 0.0
    terminated = False
    truncated = False
  Step 8
    S = 8
    A = 1
    R = 0.0
```

```
    terminated = False
    truncated = False
  Step 9
    S = 9
    A = 1
    R = 0.0
    terminated = False
    truncated = False
```

Problem:

2. Now implement tabular Q-learning and use it to learn the optimal policy on the *Frozen Lake* environment implemented above. Sample the environment ε-greedily and use a discount factor of $\lambda = 0.99$. Complete the code below and check that the learned values and policy are (close) to those computed above.

Note: Q-learning is a stochastic algorithm, and for simplicity we are using a fixed learning rate. There may be unlucky runs where the learned policy is not quite optimal, but most runs should be.

In [160]:
```
### Q-learning ###
# from tqdm import tqdm

env = FrozenLakeEnvironment()

# Hyperparameters
discount = 0.99 # Discount factor
num_episodes = 100000 # Number of episodes to learn for 25000
alpha = 0.1 # Learning rate
epsilon = 0.5 # Exploration rate for epsilon-greedy action selection

# Initialize
Q = np.zeros((env.num_states, env.num_actions))

num_actions = len(actions)

def epsilon_greedily(state):
    if np.random.uniform(0, 1) < epsilon:
        # Explore: choose a random action
        action = np.random.choice(num_actions)
    else:
        # Exploit: choose the action with the highest Q-value for the cu
        action = np.argmax(Q[state])
    return action

# Learn
for episode in range(num_episodes):
    state = env.reset()
    env.terminated = False
    env.truncated = False

    while not (env.terminated or env.truncated):
        # Pick action epsilon-greedily
        action = epsilon_greedily(state)
        # Sample environment
        next_state, reward, terminated, truncated, _ = env.step(action)
        # Update Q-table
        Q[state, action] += alpha * (reward + discount * np.max(Q[next_s
        state = next_state

print("Q-table:")
print(Q)

def extract_optimal_policy(Q):
    optimal_policy = {}
    for state in range(Q.shape[0]):
        optimal_policy[state] = action_nums[np.argmax(Q[state])]
    return optimal_policy

print(extract_optimal_policy(Q))
```

```
Reward: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]

Q-table:
[[0.22032992 0.20289816 0.2157325  0.20188471]
 [0.23255823 0.11301125 0.17951644 0.16628338]
 [0.213279   0.23574125 0.26470061 0.21087585]
 [0.18300278 0.18197655 0.11227625 0.11831123]
 [0.11952182 0.12145594 0.14592199 0.11877992]
 [0.         0.         0.         0.        ]
 [0.09557512 0.2761957  0.33772677 0.15174206]
 [0.         0.         0.         0.        ]
 [0.17376661 0.19864803 0.2274942  0.12017804]
 [0.26765101 0.44755924 0.30466297 0.51391597]
 [0.3307762  0.37852298 0.53358864 0.43304319]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.40538627 0.67345382 0.44803751 0.33445869]
 [0.71539153 0.74438799 0.68923763 0.80350842]
 [0.         0.         0.         0.        ]]
{0: 'N', 1: 'N', 2: 'W', 3: 'N', 4: 'W', 5: 'N', 6: 'W', 7: 'N', 8:
'W', 9: 'S', 10: 'W', 11: 'N', 12: 'N', 13: 'E', 14: 'S', 15: 'N'}
```