In [20]:

```
import sys
import time
from typing import Tuple, List, Dict, Any, Callable

import gym

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from frozen_lake import MDP
```

In [21]:

```
def simple_plot(
    arr,
    xlabel: str,
    ylabel: str,
    title: str,
    save_path: str = "",
    show: bool = True,
    timeout: int = None,
) -> None:
    if len(arr) == 2:
        plt.plot(arr[0], arr[1])
    else:
        plt.plot(arr)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)

    if save_path:
        plt.savefig(save_path)

    if timeout is not None:
        plt.pause(timeout)
        plt.draw()
        plt.waitforbuttonpress(timeout=5)
        plt.close()
    elif show:
        plt.show()
    plt.close()
```

In [22]:

```
# Create the environment
env = gym.make("FrozenLake-v1")
env.reset()
'''
The surface is described using a grid like the following:
SFFF       (S: starting point, safe)
FHFH       (F: frozen surface, safe)
FFFH       (H: hole, fall to your doom)
HFFG       (G: goal, where the frisbee is located)
'''
print("Observation space: ", env.observation_space)
print("Action space: ", env.action_space)
```

```
Observation space:  Discrete(16)
Action space:  Discrete(4)
```

In [23]:

```
# Create an MDP from the env as a reference
mdp = MDP(env)
```

In [24]:

```
actions = {
    'Left': 0,
    'Down': 1,
    'Right': 2,
    'Up': 3
}

act_seq = (2 * ['Right']) + (3 * ['Down'] + ['Right'])
print(f"Action sequence: {act_seq}")


env.render()

for a in act_seq:
    obs, rew, done, info = env.step(actions[a])
    env.render()
    print(f"Reward: {rew:.2f}")
    print(info, '\n')
    if done:
        break
```

```
Action sequence: ['Right', 'Right', 'Down', 'Down', 'Down', 'Right']

SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
Reward: 0.00
{'prob': 0.3333333333333333}

  (Right)
SFFF
FHFH
FFFH
HFFG
Reward: 0.00
{'prob': 0.3333333333333333}

  (Down)
SFFF
FHFH
FFFH
HFFG
Reward: 0.00
{'prob': 0.3333333333333333}

  (Down)
SFFF
FHFH
FFFH
HFFG
Reward: 0.00
{'prob': 0.3333333333333333}

  (Down)
SFFF
FHFH
FFFH
HFFG
Reward: 0.00
{'prob': 0.3333333333333333}

  (Right)
SFFF
FHFH
FFFH
HFFG
Reward: 0.00
{'prob': 0.3333333333333333}
```

1.1

Describe the environment state and action spaces, and reward function. Given a state and an action, is the state transition deterministic?

State space: $S \in \{0, 1, 2, \ldots, 15\}$ represents the index from top-left to bottom-right of 4x4 grid. Each state s can be $\{S, F, H, G\}$

where

```
S: starting point, safe
F: frozen surface, safe
H: hole, fall to your doom
G: goal, where the frisbee is located
```

The terminal state is the goal state G and hole state H.

Reward is 0 for every step taken in $\{S, F, H\}$, 1 for reaching the final goal state G.

$$r = 1 \text{ if s = G}$$

$$r = 0 \text{ Otherwise}$$

The state transition is not deterministic, becasue the transition probability of given state and action is not 1.

1.2

Starting with teh defintion of a value function, show that for a deterministic policy $\pi(s)$, the value function $v(s)$ can be expressed as:

$$v(s) = \sum_{s' \in S} p(s'|s, a)\left[r(s, a, s') + \gamma v(s')\right]$$

Return $G_t = \sum_{\tau=t}^{T} \gamma^{\tau-t} R_\tau$

Assume probabilistic transitions $T(s, a, s') = R(s'|a, s)$

Deterministic policy $\pi(s) = a \; \forall a \in A$

$$v(s) = E_\pi\left[G_t|S_t = s\right]$$

$$v(s) = E_\pi[\sum_{k=t}^{\infty} \gamma^{k-t} R_k | s_t = s]$$

$$= E_\pi[R_t + \gamma \sum_{k=t+1}^{\infty} \gamma^{k-t} R_k | s_t = s]$$

$$= \sum_a \pi(a|s) \sum_{s'} p(s'|a, s)\left[r(s, a, s') + \gamma E_\pi[\sum_{k=t}^{\infty} \gamma^{k-t} R_k | s_{t+1} = s']\right]$$

$$= \sum_a \pi(a|s) \sum_{s'} p(s'|a, s)\left[r(s, a, s') + \gamma v(s')\right]$$

In our case, $a = \pi(s)$ is a deterministic policy, we can omit the probability of policy in the above equation.

$$v(s) = \sum_{s' \in S} p(s'|s, a)\left[r(s, a, s') + \gamma v(s')\right]$$

1.3

Write a functionTestPolicy(policy), that returns the average rate of successful episodes over 100 trials for a deterministic policy. What is the success rate of a policy(number of times completed / total number of trials) given by $\pi(s) = (s + 1)$.

In [25]:

```
def TestPolicy(
    self,
    policy: Callable,
    trials: int = 100,
    render: bool = False,
    verbose: bool = False,
) -> float:
    """
    Test a policy by running it in the given environment.

    :param policy: A policy to run.
    :param env: The environment to run the policy in.
    :param render: Whether to render the environment.

    :returns: success rate over # of trials.
    """
    assert trials > 0 and isinstance(trials, int)

    success = 0
    reward = 0
    for _ in range(trials):
        obs = self.env.reset()
        done = False
        while not done:
```

```
            act = policy(obs)
            obs, rew, done, info = self.env.step(act)
            reward += rew
            if render:
                self.env.render()
                time.sleep(0.1)
            if done and obs == 15:
                success += 1
    success_rate = success / trials
    mean_reward = reward / trials
    if verbose:
        print(f"Success rate: {success_rate}")
    return success_rate, mean_reward
```

In [26]:

```
# 3. Naive policy
policy = lambda s: (s + 1) % 4
naive_success_rates = []
for _ in range(10):
    naive_success_rate, _ = mdp.TestPolicy(policy, render=False)
    naive_success_rates.append(naive_success_rate)
print(f"Average naive_success_rates: {np.mean(naive_success_rates)}")
```

```
Average naive_success_rates: 0.013000000000000001
```

1.4

Write a functionLearnModel, that returns the transition probabilities $p(s'|a, s)$ andreward function $r(s, a, s')$. Estimate these values over 10^5 random samples.

In [27]:

```
def learnModel(self, n_samples: int = 10 ** 5) -> Tuple[np.ndarray, np.ndarray]:
    """
    Estimate transition probabilities p(s'|a, s) and
    reward function r(s, a, s') over n_samples random samples.

    :param n_samples: Number of random samples to use.
    :returns: transition probabilities and reward function.
    """
    assert n_samples > 0 and isinstance(n_samples, int)

    # Dimension of observation space and action space (both discrete)
    P = np.zeros((self.nS, self.nA, self.nS))  # transition probability: S x A x S' -> [0, 1]
    R = np.zeros_like(P)  # reward r(s, a, s')

    obs = self.env.reset()
    done = False
    for _ in range(n_samples):
        # Random action
        act = env.action_space.sample()
        nxt_obs, rew, done, _ = self.env.step(act)

        P[obs, act, nxt_obs] += 1
        R[obs, act, nxt_obs] += rew
        obs = nxt_obs

        if done:
            obs = self.env.reset()
    # Normalize transition probabilities -> [0,1]
    p1 = P.copy()  # Don't modify P dircetly
    for s in range(self.nS):
        for a in range(self.nA):
            total_counts = np.sum(P[s, a, :])
            if total_counts != 0:
                p1[s, a, :] /= total_counts
    # Avoid division by zero error
    R /= np.where(P!=0, P, 1)

    # Store the estimated transition probabilities and reward function
    self.P_hat = p1
    self.R_hat = R
    return p1, R
```

In [28]:

```
p1, r1 = mdp.learnModel(n_samples=10 ** 5)
mse = lambda x, y: np.mean((x - y) ** 2)

MSE_P, MSE_R =  mse(mdp.P, p1), mse(mdp.R, r1)
print(f"Mean square error of P: {MSE_P}, \nMean square error of R: {MSE_R}")
```

```
Mean square error of P: 0.01957353902437787,
Mean square error of R: 0.0013020833333333335
```

1.5

Write a function PolicyEval() for evaluating a given deterministic policy and with the help of this function implement a policy iteration method to solve this environmentover 50 iterations. Plot the average rate of success of the learned policy at everyiteration.

In [29]:

```
def PolicyEval(
    self,
    V: np.ndarray,
    policy: np.ndarray,
    gamma: float,
    theta: float
):
    """
    Policy evaluation
    :param V: value function
    :param policy: a policy
    :param gamma: discount factor
    :param theta: tolerance or termination threshold
    """
    assert 0 < gamma < 1
    assert 0 < theta <= 1e-2, "Threshold should be a small positive number"

    # Using the esitimations of the transition probabilities and reward function
    if self.P_hat is None or self.R_hat is None:
        self.learnModel()

    while True:
        delta = 0.0
        for s in range(self.nS):
            act = policy[s]
            Vs = 0
            for nxt_s in range(self.nS):
                Vs += self.P_hat[s, act, nxt_s] * (self.R_hat[s, act, nxt_s] + gamma * V[nxt_s])
            # Calculate delta
            delta = max(delta, abs(Vs - V[s]))
            # Update V
            V[s] = Vs
        if delta < theta:
            break
    return V
```

In [30]:

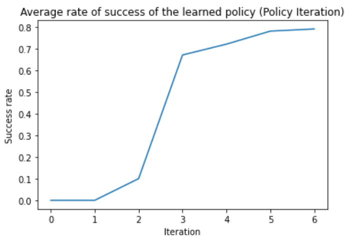
```
def PolicyIteration(
    self,
    max_iter: int = 50,
    gamma: float = 0.99,
    theta: float = 1e-8
):
    """
    Policy iteration
    :param policy: a policy
    :param max_iter: maximum number of iterations
    :param gamma: discount factor
    :param theta: tolerance or termination threshold
    """
    assert max_iter > 0 and isinstance(max_iter, int)
    assert 0 < gamma < 1
    assert 0 < theta <= 1e-2, "Theta should be a small positive number"

    # Initialize V(s), \pi(s)
    V = np.zeros(self.nS)
    PI = np.zeros(self.nS, dtype=int)  # since actions are integers
    success_rates = []
    mean_rewards = []

    print(f'\n-------- Policy Iteration --------:')
    for i in range(max_iter):
        PI_old = PI.copy()
        print(f'Iteration {i+1}: ', end='')
        # Policy Evaluation
        V = self.PolicyEval(V, PI, gamma, theta)

        # Policy Improvement
        PI = self.PolicyImprovement(V, gamma)

        PI_fn = lambda s: PI[s]
        success_rate, mean_rew = self.TestPolicy(PI_fn, trials=100, render=False, verbose=True)
        success_rates.append(success_rate)
        mean_rewards.append(mean_rew)

        if np.all(PI_old == PI):
            print(f"\nPolicy is stable in {i} iterations")
            break

    return PI, V, success_rates, mean_rewards
```

Policy Iteration Converge less than 10 iterations.

In [31]:

```
# 5. Policy iteration
PI, V_pi, success_rates, mean_rewards = mdp.PolicyIteration(50, theta=sys.float_info.epsilon, exhaustive=False)
print(f"PI: {PI}")
print(f"V_pi: {V_pi}")

plt.plot(success_rates)
plt.xlabel("Iteration")
plt.ylabel("Success rate")
plt.title("Average rate of success of the learned policy (Policy Iteration)")
plt.show()
```

```
-------- Policy Iteration --------:
Iteration 1: Success rate: 0.0
Iteration 2: Success rate: 0.0
Iteration 3: Success rate: 0.1
Iteration 4: Success rate: 0.67
Iteration 5: Success rate: 0.72
Iteration 6: Success rate: 0.78
Iteration 7: Success rate: 0.79

Policy is stable in 6 iterations
PI: [0 3 3 3 0 0 0 0 3 1 0 0 0 2 1 0]
V_pi: [0.51276631 0.47393719 0.44811929 0.43499381 0.52815534 0.
 0.3550636  0.         0.55867053 0.60933542 0.59566271 0.
 0.         0.7049711  0.83785129 0.        ]
```



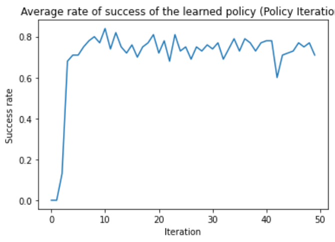
Policy Iteration exhaust all 50 iterations.

In [32]:

```
PI, V_pi, success_rates, mean_rewards = mdp.PolicyIteration(50, theta=sys.float_info.epsilon, exhaustive=True)
print(f"PI: {PI}")
print(f"V_pi: {V_pi}")

plt.plot(success_rates)
plt.xlabel("Iteration")
plt.ylabel("Success rate")
plt.title("Average rate of success of the learned policy (Policy Iteration)")
plt.show()
```

```
-------- Policy Iteration --------:
Iteration 1: Success rate: 0.0
Iteration 2: Success rate: 0.0
Iteration 3: Success rate: 0.13
```

```
Iteration 4: Success rate: 0.68
Iteration 5: Success rate: 0.71
Iteration 6: Success rate: 0.71
Iteration 7: Success rate: 0.75
Iteration 8: Success rate: 0.78
Iteration 9: Success rate: 0.8
Iteration 10: Success rate: 0.77
Iteration 11: Success rate: 0.84
Iteration 12: Success rate: 0.74
Iteration 13: Success rate: 0.82
Iteration 14: Success rate: 0.75
Iteration 15: Success rate: 0.72
Iteration 16: Success rate: 0.76
Iteration 17: Success rate: 0.7
Iteration 18: Success rate: 0.75
Iteration 19: Success rate: 0.77
Iteration 20: Success rate: 0.81
Iteration 21: Success rate: 0.72
Iteration 22: Success rate: 0.78
Iteration 23: Success rate: 0.68
Iteration 24: Success rate: 0.81
Iteration 25: Success rate: 0.73
Iteration 26: Success rate: 0.75
Iteration 27: Success rate: 0.69
Iteration 28: Success rate: 0.75
Iteration 29: Success rate: 0.73
Iteration 30: Success rate: 0.76
Iteration 31: Success rate: 0.74
Iteration 32: Success rate: 0.77
Iteration 33: Success rate: 0.69
Iteration 34: Success rate: 0.74
Iteration 35: Success rate: 0.79
Iteration 36: Success rate: 0.73
Iteration 37: Success rate: 0.79
Iteration 38: Success rate: 0.77
Iteration 39: Success rate: 0.73
Iteration 40: Success rate: 0.77
Iteration 41: Success rate: 0.78
Iteration 42: Success rate: 0.78
Iteration 43: Success rate: 0.6
Iteration 44: Success rate: 0.71
Iteration 45: Success rate: 0.72
Iteration 46: Success rate: 0.73
Iteration 47: Success rate: 0.77
Iteration 48: Success rate: 0.75
Iteration 49: Success rate: 0.77
Iteration 50: Success rate: 0.71
PI: [0 3 3 3 0 0 0 0 3 1 0 0 0 2 1 0]
V_pi: [0.51276631 0.47393719 0.44811929 0.43499381 0.52815534 0.
 0.3550636  0.         0.55867053 0.60933542 0.59566271 0.
 0.         0.7049711  0.83785129 0.        ]
```



1.6

Write a function ValueIter() that returns a deterministic policy learned through value-iteration over 50 iterations. Plot the average rate of success of the learned policy atevery iteration.

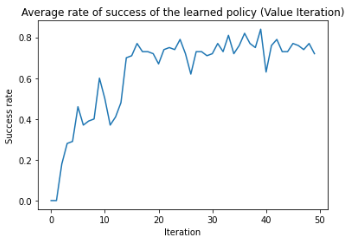
In [33]:

```python
def ValueIter(
    self,
    max_iter: int = 50,
    gamma: float = 0.99,
    theta: float = 1e-8
):
    assert max_iter > 0 and isinstance(max_iter, int)
    assert 0 < gamma < 1

    # Initialize V(s), \pi(s)
    V = np.zeros(self.nS)
    PI = np.zeros(self.nS, dtype=int)  # since actions are integers
    success_rates = []
    mean_rewards = []

    print(f'\n-------- Value Iteration --------:')
    for i in range(max_iter):
        print(f'Iteration {i+1}: ', end='')
        delta = 0.0
        for s in range(self.nS):
            v_old = V[s]
            # V(s) = max_a Q(s, a)
            Q = self.qValue(V, s, gamma)  # Q(s_t, a) -> Vector of Q-values
            V[s] = max(Q)

            delta = max(delta, abs(V[s] - v_old))
        if delta < theta:
            break


        PI = self.PolicyImprovement(V, gamma)

        PI_fn = lambda s: PI[s]
        success_rate, mean_rew = self.TestPolicy(PI_fn, trials=100, render=False, verbose=True)
        success_rates.append(success_rate)
        mean_rewards.append(mean_rew)
    return PI, V, success_rates, mean_rewards
```

In [34]:

```python
# 6. Value iteration
PI, V_pi, success_rates, mean_rewards = mdp.ValueIter(50, theta=sys.float_info.epsilon)
print(f"PI: {PI}")
print(f"V_pi: {V_pi}")

plt.plot(success_rates)
plt.xlabel("Iteration")
plt.ylabel("Success rate")
plt.title("Average rate of success of the learned policy (Value Iteration)")
plt.show()
```

```
-------- Value Iteration --------:
Iteration 1: Success rate: 0.0
Iteration 2: Success rate: 0.0
Iteration 3: Success rate: 0.18
Iteration 4: Success rate: 0.28
Iteration 5: Success rate: 0.29
Iteration 6: Success rate: 0.46
Iteration 7: Success rate: 0.37
Iteration 8: Success rate: 0.39
Iteration 9: Success rate: 0.4
Iteration 10: Success rate: 0.6
Iteration 11: Success rate: 0.5
Iteration 12: Success rate: 0.37
Iteration 13: Success rate: 0.41
Iteration 14: Success rate: 0.48
Iteration 15: Success rate: 0.7
Iteration 16: Success rate: 0.71
Iteration 17: Success rate: 0.77
Iteration 18: Success rate: 0.73
Iteration 19: Success rate: 0.73
Iteration 20: Success rate: 0.72
Iteration 21: Success rate: 0.67
Iteration 22: Success rate: 0.74
Iteration 23: Success rate: 0.75
Iteration 24: Success rate: 0.74
Iteration 25: Success rate: 0.79
Iteration 26: Success rate: 0.72
Iteration 27: Success rate: 0.62
Iteration 28: Success rate: 0.73
Iteration 29: Success rate: 0.73
Iteration 30: Success rate: 0.71
Iteration 31: Success rate: 0.72
Iteration 32: Success rate: 0.77
Iteration 33: Success rate: 0.73
Iteration 34: Success rate: 0.81
Iteration 35: Success rate: 0.72
Iteration 36: Success rate: 0.76
Iteration 37: Success rate: 0.82
Iteration 38: Success rate: 0.77
Iteration 39: Success rate: 0.75
Iteration 40: Success rate: 0.84
Iteration 41: Success rate: 0.63
Iteration 42: Success rate: 0.76
Iteration 43: Success rate: 0.79
Iteration 44: Success rate: 0.73
Iteration 45: Success rate: 0.73
Iteration 46: Success rate: 0.77
Iteration 47: Success rate: 0.76
Iteration 48: Success rate: 0.74
Iteration 49: Success rate: 0.77
Iteration 50: Success rate: 0.72
PI: [0 3 3 3 0 0 0 0 3 1 0 0 0 2 1 0]
V_pi: [0.44548399 0.3938936  0.35943487 0.34182928 0.46805345 0.
 0.3156702  0.         0.50882757 0.57310608 0.566467   0.
 0.         0.67864816 0.82399567 0.        ]
```



2.1. Solve the environment using Q-learning over 5000 episodes. For exploration duringtraining, take random actions with probability 1-e/5000 where e is the number ofcurrent episode. Plot the success rate of the learned policy at an interval of 100episodes.

(a) Train the policy using the following learning rates with γ= 0.99.Report what you observe.

A small α tends to work better than

With a larger alpha, the Q Learning will add more weight to the reward to go. However, Q-Learning learn slowly. The Q value is not very good during the early stage of learning. (agent likely not visited every state in the enviroment)

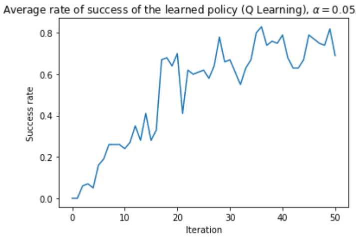
Q value may change dramatically with a large alpha which leard to instability depend on the method. MC is an unbiased estimator of the Q value but high variance. On the other hand, TD(0) is biased but low variance. Takeing a larger step size will let agent learn faster but create more variace on the estimation.

In [35]:

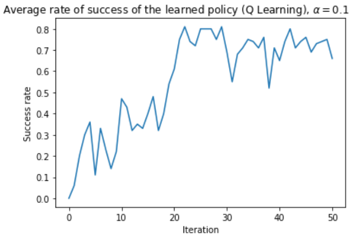
```python
# 7
learning_rate = [0.05, 0.1, 0.25, 0.5]
discount_factor = [0.9, 0.95, 0.99]
    # (a)
for lr in learning_rate:
    pi_QL, Q_pi, success_rates_QL = mdp.QLearning(
        n_episodes=5_000,
        gamma=0.99,
        alpha=lr,
        strategy="epsilon",
        verbose=False,
    )
    simple_plot(
        success_rates_QL,
        xlabel="Iteration",
        ylabel="Success rate",
        title=(
            f"Average rate of success of the learned policy (Q Learning), "
            + r"$\alpha=$"
            + f"{lr}, "
            + r"$\gamma=$"
            + f"{0.99}"
        ),
```

```
        # save_path=plot_dir / f'QL_a_{lr}_r_{g}.png',
        show=True,
    )
    print("\n")
```

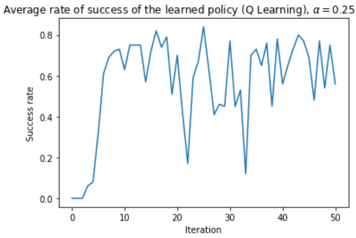
```
-------- Q Learning (alpha=0.05, gamma=0.99, strategy: epsilon) --------:
```



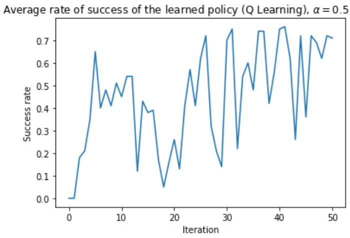
```
-------- Q Learning (alpha=0.1, gamma=0.99, strategy: epsilon) --------:
```



```
-------- Q Learning (alpha=0.25, gamma=0.99, strategy: epsilon) --------:
```



```
-------- Q Learning (alpha=0.5, gamma=0.99, strategy: epsilon) --------:
```



(b) Train the policy using the following discount factors with α= 0.05. Report what you observe.

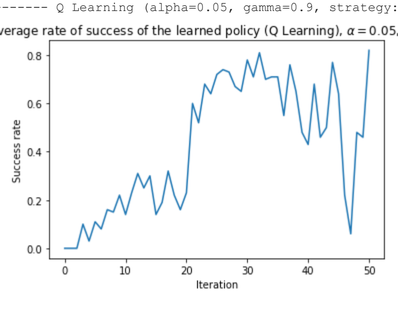
In FrozenLake-v0, the reward is 1 only in the goal state and 0 in all other states. It is better to encourage the agent to be foresight. As the discount factor γ increase , agent will focuse more on the long-term reward as opposed to the immediate reward which is the case in FrozenLake-v0.

In [36]:

```
# (b)
for g in discount_factor:
    pi_QL, Q_pi, success_rates_QL = mdp.QLearning(
        n_episodes=5_000,
        gamma=g,
        alpha=0.05,
        strategy="epsilon",
        verbose=False,
    )
    simple_plot(
        success_rates_QL,
        xlabel="Iteration",
        ylabel="Success rate",
        title=(
            f"Average rate of success of the learned policy (Q Learning), "
            + r"$\alpha=$"
            + f"{0.05}, "
            + r"$\gamma=$"
            + f"{g}"
        ),
        # save_path=plot_dir / f'QL_a_{lr}_r_{g}.png',
        show=True,
    )
    print("\n")
```

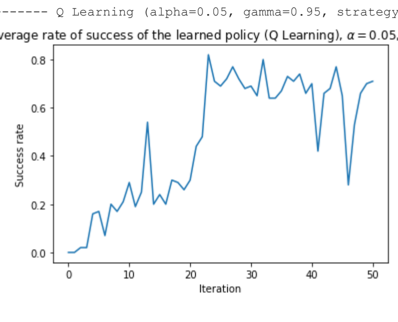
-------- Q Learning (alpha=0.05, gamma=0.9, strategy: epsilon) --------:

Average rate of success of the learned policy (Q Learning), $\alpha = 0.05$, $\gamma = 0.9$



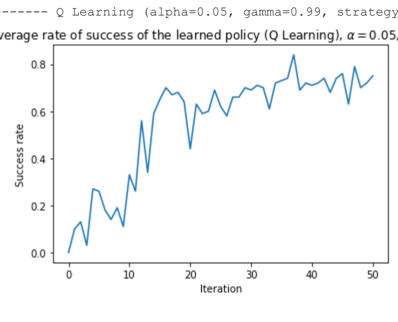
-------- Q Learning (alpha=0.05, gamma=0.95, strategy: epsilon) --------:

Average rate of success of the learned policy (Q Learning), $\alpha = 0.05$, $\gamma = 0.95$



-------- Q Learning (alpha=0.05, gamma=0.99, strategy: epsilon) --------:

Average rate of success of the learned policy (Q Learning), $\alpha = 0.05$, $\gamma = 0.99$



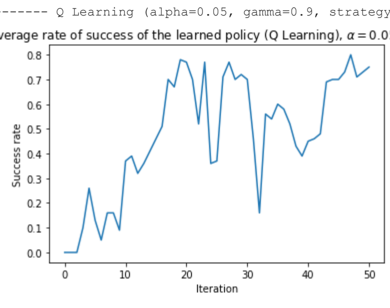
In the previous question, the exploration was linearly annealed. Solve the environmentusing Q-learning by proposing a different strategy to explore. Find a suitable α and γ for your method. Report your strategy and training results.

In [37]:

```
[illegible]
```

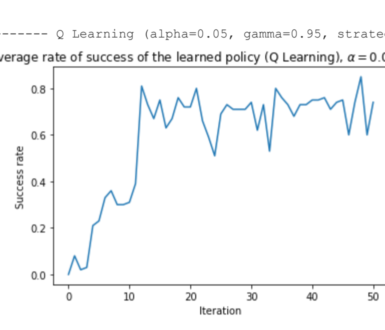
-------- Q Learning (alpha=0.05, gamma=0.9, strategy: exponential) --------:

Average rate of success of the learned policy (Q Learning), $\alpha = 0.05$, $\gamma = 0.9$



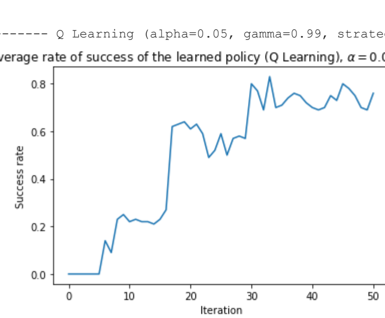
-------- Q Learning (alpha=0.05, gamma=0.95, strategy: exponential) --------:

Average rate of success of the learned policy (Q Learning), $\alpha = 0.05$, $\gamma = 0.95$



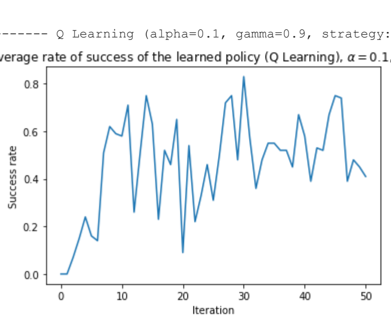
-------- Q Learning (alpha=0.05, gamma=0.99, strategy: exponential) --------:

Average rate of success of the learned policy (Q Learning), $\alpha = 0.05$, $\gamma = 0.99$



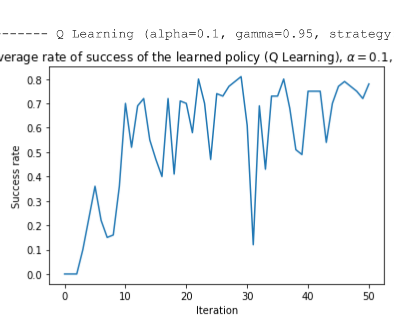
-------- Q Learning (alpha=0.1, gamma=0.9, strategy: exponential) --------:

Average rate of success of the learned policy (Q Learning), $\alpha = 0.1$, $\gamma = 0.9$



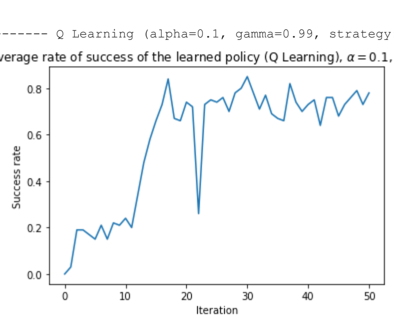
-------- Q Learning (alpha=0.1, gamma=0.95, strategy: exponential) --------:

Average rate of success of the learned policy (Q Learning), $\alpha = 0.1$, $\gamma = 0.95$



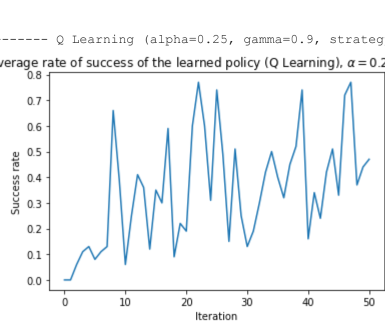
-------- Q Learning (alpha=0.1, gamma=0.99, strategy: exponential) --------:

Average rate of success of the learned policy (Q Learning), $\alpha = 0.1$, $\gamma = 0.99$



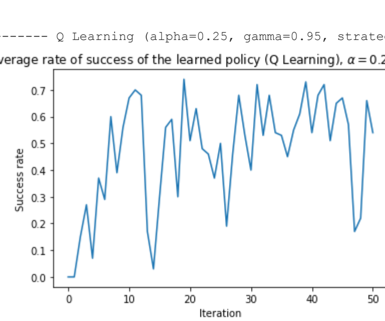
-------- Q Learning (alpha=0.25, gamma=0.9, strategy: exponential) --------:

Average rate of success of the learned policy (Q Learning), $\alpha = 0.25$, $\gamma = 0.9$



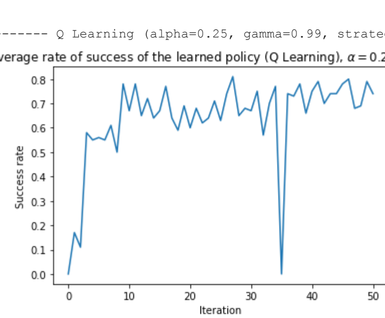
-------- Q Learning (alpha=0.25, gamma=0.95, strategy: exponential) --------:

Average rate of success of the learned policy (Q Learning), $\alpha = 0.25$, $\gamma = 0.95$



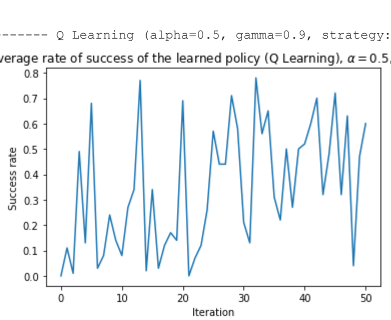
-------- Q Learning (alpha=0.25, gamma=0.99, strategy: exponential) --------:

Average rate of success of the learned policy (Q Learning), $\alpha = 0.25$, $\gamma = 0.99$



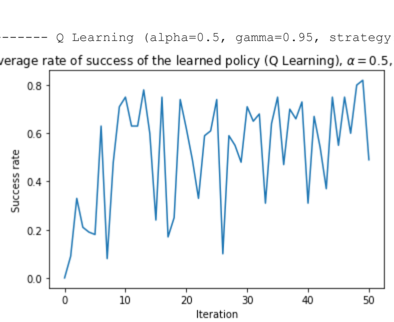
-------- Q Learning (alpha=0.5, gamma=0.9, strategy: exponential) --------:

Average rate of success of the learned policy (Q Learning), $\alpha = 0.5$, $\gamma = 0.9$



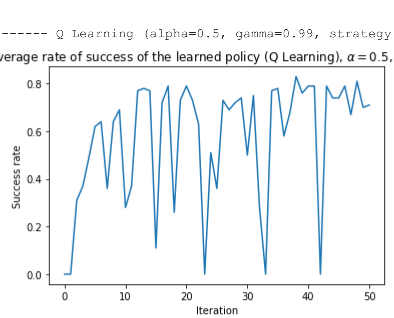
-------- Q Learning (alpha=0.5, gamma=0.95, strategy: exponential) --------:

Average rate of success of the learned policy (Q Learning), $\alpha = 0.5$, $\gamma = 0.95$



-------- Q Learning (alpha=0.5, gamma=0.99, strategy: exponential) --------:

Average rate of success of the learned policy (Q Learning), $\alpha = 0.5$, $\gamma = 0.99$



A simple Count-based Exploration by using exponential decay:

$$\frac{dN}{dt} = -\lambda N$$

The solution to this equation is:

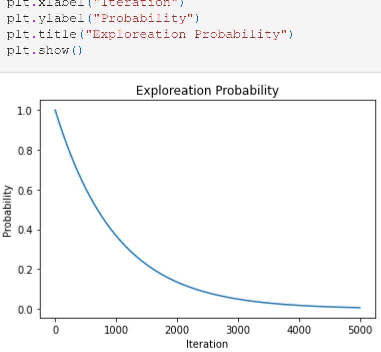
$$N(t) = N_0 e^{-\lambda t}$$

In this case N_0 is the initial exloration probability at $t = 0$ is 1 and the constatnt λ is the decay constant, $\lambda = 0.001$. Therefore, agent will absolutaly explore at $t = 0$ and exponentially decay the exploration probability across time.

This exploration strategy does not work well in hard exploration problem but it is good enough for our environment.

In [38]:

```
prob = []
init_prob = 1
lamd = 0.001
for t in range(5_000):
    Nt = init_prob * np.exp(-lamd * t)
    prob.append(Nt)

plt.plot(prob)
plt.xlabel("Iteration")
plt.ylabel("Probability")
plt.title("Exploreation Probability")
plt.show()
```



In [ ]: