目录

Task 1 Frozen Lake MDP

Policy Iteration

1. Policy Evaluation
   固定当前策略π，更新状态价值函数V直至收敛。

```
# 反复迭代直至收敛
while True:
        value_function_next = np.zeros_like(value_function)
        for s in range(nS):  # 一轮迭代
                a = policy[s]
                prob, nextstate, reward, done = P[s][a][0]  # 此prob为π(s,a
                value_function_next[s] = prob * (reward + gamma*value_funct
        if np.max(np.abs(value_function_next - value_function)) < tol:
                break
        value_function = value_function_next
```

2. Policy Improvement

贪心选取使q(s, a)最大的a作为s的策略动作。

```
# 确定性动作
for s in range(nS):
        q = np.zeros(nA)
        for a in range(nA):
                prob, nextstate, reward, done = P[s][a][0]  # 此prob为π(s,a
                q[a] = prob * (reward + gamma * value_from_policy[nextstate
        new_policy[s] = np.argmax(q)
```

3. 重复策略评估和策略改进，直至策略收敛

```
while True:
        value_function = policy_evaluation(P, nS, nA, policy, gamma, tol)
        new_policy = policy_improvement(P, nS, nA, value_function, None, ga
        if (new_policy == policy).all():
                break
        else:
                policy = new_policy
```

4. 最终的V值和策略分别如下图：

```
SFFF
FHFH
FFFH
HFFG
Episode reward: 1.000000
```

: 0	: 1	: 2	: 3	: 4	: 5	: 6	: 7	: 8	: 9	: 10	: 11	: 12	: 13	: 14	: 15
0.59049	0.65610	0.72900	0.65610	0.65610	0.00000	0.81000	0.00000	0.72900	0.81000	0.90000	0.00000	0.00000	0.90000	1.00000	0.00000

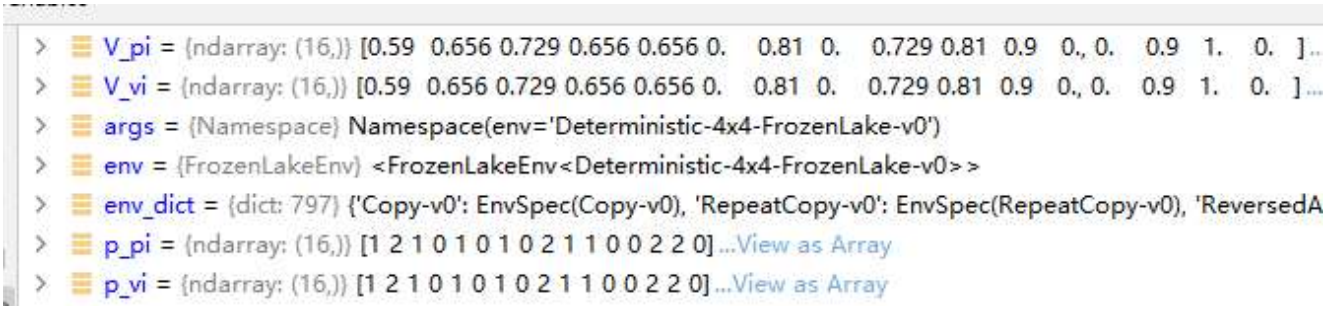
: 0	: 1	: 2	: 3	: 4	: 5	: 6	: 7	: 8	: 9	: 10	: 11	: 12	: 13	: 14	: 15
1	2	1	0	1	0	1	0	2	1	1	0	0	2	2	0

Value Iteration

迭代更新最优值函数直至收敛，并执行一次策略提取。

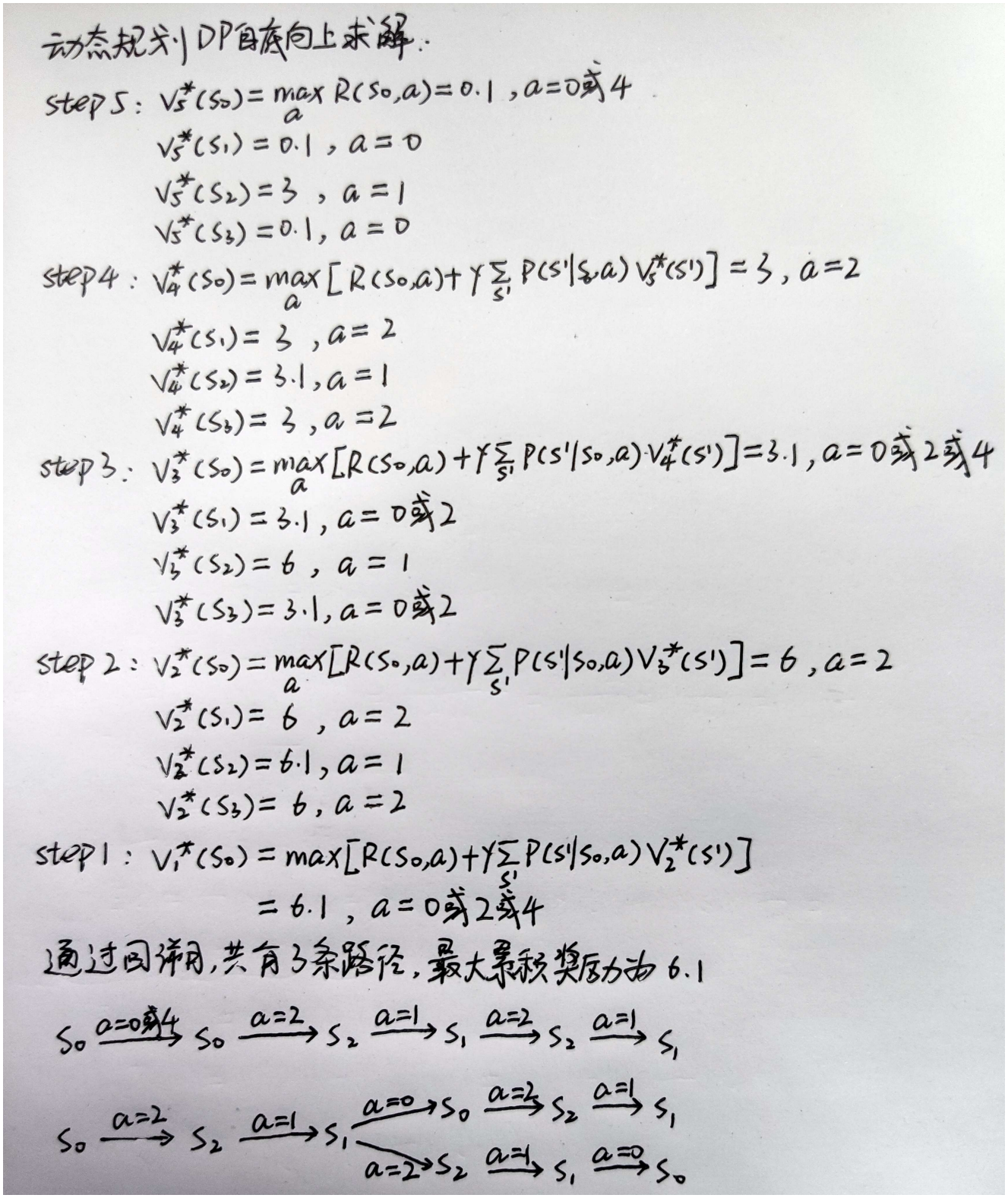
```
while True:
        value_function_next = np.zeros_like(value_function)
        for s in range(nS):
                q = np.zeros(nA)
                for a in range(nA):
                        prob, nextstate, reward, done = P[s][a][0]  # 此prob为π(
                        q[a] = prob * (reward + gamma * value_function[nextstate
                        value_function_next[s] = np.max(q)
        if np.max(np.abs(value_function_next - value_function)) < tol:
                break
        value_function = value_function_next

# 一次策略提取
for s in range(nS):
        q = np.zeros(nA)
        for a in range(nA):
                prob, nextstate, reward, done = P[s][a][0]  # 此prob为π(s,a,s'),
                q[a] = prob * (reward + gamma * value_function[nextstate])
        policy[s] = np.argmax(q)
```

V值和策略P与策略迭代方法相同。



Task 2 Test Environment

使用动态规划DP自底向上求解，如下图，因为DP是精确算法，所以求解的是最优解。



Task 3 Tabular Q-Learning

agent探索概率epsilon线性递减

```
assert self.nsteps >= 1
k = (self.eps_end - self.eps_begin) / (self.nsteps - 0)
if t <= self.nsteps:
    self.epsilon = self.eps_begin + k * (t - 0)
else:
    self.epsilon =  self.eps_end
```

$\epsilon - \mathbf{greedy}$ Exploration Strategy

```
if np.random.rand() <= self.epsilon:
    return self.env.action_space.sample()
else:
    return best_action
```

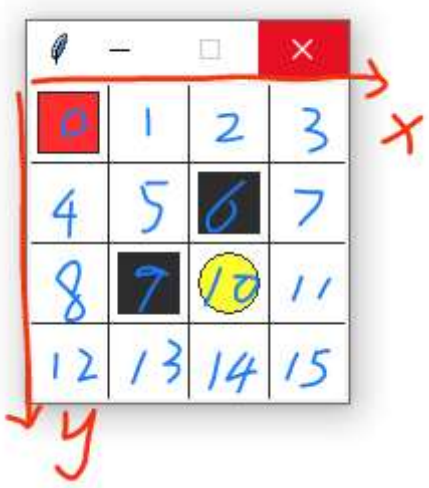
Task 4 Maze Example

实现Q Learning和SARSA算法，两个算法很类似，只不过SARSA是on policy，其用于产生训练数据的决策策略和更新Q表的优化策略相同，都是$\epsilon - \mathbf{greedy}$，Q Learning是off policy，其决策策略使用$\epsilon - \mathbf{greedy}$，优化策略使用贪心策略。

1. 观测值说明

agent的观测值(x1, y1, x2, y2)，(x1, y1)为agent所处的左上角坐标，(x2, y2)为agent的右下角坐标。将16个左上角坐标映射为0-15，作为状态，也即Q表的行索引，已标记在下图中。坐标系即状态空间如下图所示。由左上角左边换算为Q标行索引的公式为

```
index = (x1-5)//40 + (y1-5)//40 * 4
```



2. 动作空间说明

由environment的实现逻辑，在4x4的网格内，0：向上移动一格，1：向下移动一格，2：向右移动一格，3：向左移动一格；当执行动作会超出边界时，则保存在原位置不动。

3. Q Learning实验

部分核心代码如下：
初始化Q表

```
np.random.seed(1)
obs_dim = 4 * 4  # 观测空间为16
self.q = np.random.rand(obs_dim, len(actions))
```

选择动作

```
# epsilon 贪心
if np.random.rand() < self.epsilon:
    action = np.random.choice(len(self.actions))
else:
    # 由观测坐标换算为状态序号，agent左上角坐标为(x=observation[0], y=observation[1])
    # 换算公式为index = (x-5)//40 + (y-5)//40 * 4
    x, y = int(observation[0]), int(observation[1])  # 提取observation的左上角坐标
    s_idx = (x-5)//40 + (y-5)//40 * 4                # 换算为Q表的行下标
    action = np.argmax(self.q[s_idx, :])
return action
```

更新Q表

```
x, y = int(s[0]), int(s[1])  # 提取s的左上角坐标
idx_s = (x - 5) // 40 + (y - 5) // 40 * 4  # 换算为Q表的行下标
if self.check_state_exist(s_):
    td_target = r + self.gamma * 0  # 终止状态的Q值为0
else:
    x_, y_ = int(s_[0]), int(s_[1])  # 提取s_的左上角坐标
    idx_s_ = (x_ - 5) // 40 + (y_ - 5) // 40 * 4  # 换算为Q表的行下标
    td_target = r + self.gamma * np.max(self.q[idx_s_, :])
self.q[idx_s, a] = self.q[idx_s, a] + self.lr * (td_target - self.q[idx_s, a])
```

判断终止状态

```
if state == 'terminal':
    return True
else:
    return False
```

ϵ设为0.2，训练500个episodes，Q表如下，结合上图的坐标系、状态空间和动作空间，可以发现已经学习到策略：在状态5，Q(5,1)和Q(5,2)很小，能避免黑块；而在黄色奖励附近，Q(11,3)和Q(14,0)很大，能够到达奖励。可视化实验也显示agent总是能较快到达奖励位置。

	0	1	2	3
0	0.26103	0.27110	0.22997	0.26097
1	0.21489	0.17712	0.51203	0.28064
2	0.41524	0.20901	0.61461	0.43132
3	0.23225	0.66207	0.06407	0.65754
4	0.27985	0.28173	0.32101	0.28214
5	0.39995	-0.05496	-0.32341	0.29888
6	0.87639	0.89461	0.08504	0.03905
7	0.25104	0.68367	0.14660	0.27237
8	0.28671	0.28670	-0.41107	0.28641
9	0.68650	0.83463	0.01829	0.75014
10	0.98886	0.74817	0.28044	0.78928
11	0.28949	0.44670	0.52658	0.83878
12	0.28759	0.24258	0.25921	0.28784
13	0.15225	0.27349	0.56317	0.07789
14	0.78498	0.18512	0.58149	0.57750
15	0.23251	0.42998	0.44178	0.47256

训练1000个episodes后的Q表

	0	1	2	3
0	0.27577	0.38537	0.29400	0.26904
1	0.27954	0.23148	0.57933	0.28694
2	0.44490	-0.12349	0.66924	0.44115
3	0.32283	0.77358	0.20174	0.63520
4	0.28734	0.27743	0.44126	0.30429
5	0.50362	-0.32171	-0.50948	0.31825
6	0.87639	0.89461	0.08504	0.03905
7	0.34800	0.88287	0.28027	0.03028
8	0.30216	0.28670	-0.42856	0.28613
9	0.68650	0.83463	0.01829	0.75014
10	0.98886	0.74817	0.28044	0.78928
11	0.37426	0.44732	0.58243	0.99563
12	0.28739	0.24258	0.25921	0.28755
13	0.15225	0.27349	0.56482	0.07970
14	0.82413	0.18512	0.57655	0.57611
15	0.23897	0.43006	0.44203	0.52880

4. SARSA实验

SARSA与Q Learning类似，只是SARSA更新Q表的优化策略仍是$\epsilon-\mathbf{greedy}$，为on policy算法。
与Q Learning的区别部分如下：

```
x, y = int(s[0]), int(s[1])  # 提取s的左上角坐标
idx_s = (x - 5) // 40 + (y - 5) // 40 * 4  # 换算为Q表的行下标
if self.check_state_exist(s_):
    td_target = r + self.gamma * 0  # 终止状态的Q值为0
else:
    x_, y_ = int(s_[0]), int(s_[1])  # 提取s_的左上角坐标
    idx_s_ = (x_ - 5) // 40 + (y_ - 5) // 40 * 4  # 换算为Q表的行下标
    if np.random.rand() < self.epsilon:
        action = np.random.choice(len(self.actions))
        td_target = r + self.gamma * (self.q[idx_s_, action])
    else:
        td_target = r + self.gamma * np.max(self.q[idx_s_, :])
self.q[idx_s, a] = self.q[idx_s, a] + self.lr * (td_target - self.q[idx_s, a])
```

ϵ仍设为0.2，500个episodes后的Q表如下，实验显示agent已经学到较快到达奖励的策略。

	0	1	2	3
0	0.21562	0.22275	0.20484	0.21622
1	0.21107	0.17287	0.44461	0.25230
2	0.41035	0.23356	0.50372	0.43304
3	0.24561	0.56073	0.06926	0.55808
4	0.22739	0.22526	0.25705	0.22964
5	0.34752	-0.09219	-0.22897	0.26633
6	0.87639	0.89461	0.08504	0.03905
7	0.23445	0.64223	0.14510	0.23458
8	0.23936	0.23898	-0.27268	0.23464
9	0.68650	0.83463	0.01829	0.75014
10	0.98886	0.74817	0.28044	0.78928
11	0.24567	0.44192	0.44970	0.87833
12	0.23565	0.19347	0.16900	0.23560
13	0.09577	0.27116	0.52357	0.07534
14	0.77616	0.17929	0.57556	0.57150
15	0.15810	0.42698	0.43667	0.47124

训练1000个episodes后的Q表如下。

	0	1	2	3
0	0.22335	0.22686	0.34336	0.22411
1	0.24619	0.19361	0.43041	0.24842
2	0.41470	-0.10569	0.53396	0.40584
3	0.30852	0.63538	0.17673	0.53214
4	0.22604	0.21448	0.26063	0.22962
5	0.35253	-0.17900	-0.30967	0.26077
6	0.87639	0.89461	0.08504	0.03905
7	0.29990	0.79343	0.26028	-0.02022
8	0.23733	0.23734	-0.27268	0.23445
9	0.68650	0.83463	0.01829	0.75014
10	0.98886	0.74817	0.28044	0.78928
11	0.29254	0.44087	0.52378	0.99792
12	0.23502	0.19347	0.17202	0.23454
13	0.09577	0.27116	0.52730	0.07534
14	0.81507	0.19502	0.57288	0.56954
15	0.17996	0.42754	0.43658	0.50800