

## < 지능제어 2차 숙제 결과 보고서 >

2018142102 흥의진

### 1. "agent.py"의 "Write your code here" 부분 code

#### 1.1. "agent.py"의 "Policy Evaluation" 부분 code

```
30 def policy_evaluation(self, iter, env, policy, discount=1.0):
31     HEIGHT, WIDTH = env.size()
32     new_state_values = np.zeros((HEIGHT, WIDTH))
33     iteration = 0
34     # threshold 값을 1e-3으로 설정
35     theta = 1e-3
36     # 기존 state value function에 해당하는 old_state_values 행렬을 self.values에 저장된 행렬로 초기화
37     old_state_values = self.values.copy()
38     # Policy Evaluation Loop 를 설정
39     while True:
40         # 모든 state에 대하여 value function을 계산 (full backup)
41         for height in range(HEIGHT):
42             for width in range(WIDTH):
43                 if [height, width] in env.obstacles:    # 현재 state가 obstacle의 위치에 해당된다면
44                     new_state_values[height, width] = 0 # state value 값은 0으로 설정한다
45                 else:   # 현재 state가 obstacle의 위치가 아닌 경우
46                     # 각 action에 따른 future return의 기댓값을 저장하는 행렬 action_sum을 초기화
47                     action_sum = np.zeros(4)
48                     for action in range(len(ACTIONS)): # 각 action에 대해 for loop 반복
49                         # 현재 state, action이 주어졌을 때 next state, return을 얻는다
50                         [next_height, next_width], r = env.interaction([height, width], ACTIONS[action])
51                         # 현재 action에 해당하는 policy(a|s) * action value function 값을 action_sum 행렬에 저장
52                         action_sum[action] = policy[height, width, action] * 1.0 * \
53                                         (r + (discount * old_state_values[next_height, next_width]))
54                         # 새로운 state value function을 앞서 구한 특정 state에서의 future return 기댓값들의 합으로 구함
55                         new_state_values[height, width] = np.sum(action_sum)
56             # delta 값을 모든 state에서의 value function 변화량 중 가장 큰 값으로 설정
57             delta = abs(old_state_values - new_state_values).max()
58             if delta < theta: # delta 값이 threshold 값보다 작을 경우 Policy Evaluation Loop 반복 종료
59                 break
60             iteration += 1 # iteration flag 값은 1만큼 증가
61             # 기존 state value function에 해당하는 old_state_values를 new_state_values 값들로 update
62             old_state_values = new_state_values.copy()
63             print(f"policy iteration #: {iter}, policy evaluation iteration #: {iteration}")
64             draw_value_image(iter, np.round(new_state_values, decimals=2), env=env)
65             return new_state_values, iteration
```

< 그림 1. "agent.py"의 "Policy Evaluation" 부분 code >

## 1.2. “agent.py”의 “Policy Improvement” 부분 code

```
71 def policy_improvement(self, iter, env, state_values, old_policy, discount=1.0):
72     HEIGHT, WIDTH = env.size()
73     policy = old_policy.copy()
74     # policy가 더 이상 업데이트되지 않는지를 판별하는 flag 변수인 policy_stable 값을 True로 초기화
75     policy_stable = True
76     # 모든 state에 대해 policy improvement 수행
77     for height in range(HEIGHT):
78         for width in range(WIDTH):
79             # action에 따른 future return의 값을 저장하기 위한 행렬 action_value를 초기화
80             action_value = np.zeros(4)
81             # 현재 state에서 취할 수 있는 모든 action들에 대해 반복
82             for action in range(len(ACTIONS)):
83                 # 현재 state, action이 주어졌을 때 next state, return을 얻는다
84                 [next_height, next_width], r = env.interaction([height, width], ACTIONS[action])
85                 # 현재 state, action에서의 action value function 값을 action_value 행렬에 저장
86                 action_value[action] = 1.0 * (r + discount*state_values[next_height, next_width])
87                 # action value function이 최대로 나타나는 action들을 찾아냄
88                 indices = np.argwhere(action_value == np.amax(action_value))
89                 # 현재 state에서 각 action에 대하여 greedy하게 improve된 policy를 선언해 준다
90                 for action in range(len(ACTIONS)):
91                     # 현재 action이 argmax에 해당되는 액션인 경우
92                     if action in indices:
93                         # 확률값 policy(a|s)를 1/(argmax action들의 개수)로 부여한다
94                         policy[height, width, action] = 1./len(indices)
95                     # 현재 action이 argmax에 해당되는 액션이 아닌 경우
96                     else:
97                         # 확률값 policy(a|s)를 0으로 부여한다
98                         policy[height, width, action] = 0
99                 # 만약 policy의 개선이 있다면
100                if np.all(old_policy == policy) == False:
101                    # policy_stable flag값을 False로 설정
102                    policy_stable = False
103                print('policy stable {}'.format(policy_stable))
104                draw_policy_image(iter, np.round(policy, decimals=2), env=env)
105                return policy, policy_stable
```

< 그림 2. “agent.py”의 “Policy Improvement” 부분 code >

## 2. “agent.py”의 “Write your code here” 부분에 대한 자세한 주석

### 2.1. “agent.py”의 “Policy Evaluation” 부분에 대한 자세한 주석

Line 35: Theta라는 특정 threshold를 설정하여, state value들의 update 시 차이에 해당하는 값들이 이 threshold 값보다 작을 경우 evaluation loop을 중단한다. 이를 위해 theta 값은 1e-3으로 설정했다.

Line 37: 기존 state value들을 담고 있는 행렬 old\_state\_values 변수를 생성한 후, 초기값은 self.values에 저장된 이전 policy evaluation을 통해 구한 value function값으로 설정한다. 이를 통해 policy evaluation의 수렴 속도가 더 빨라질 수 있다.

Line 39: Threshold theta보다 policy update 변화량의 차이가 더 작을 때까지 bellman equation을 통한 policy update를 반복하는 loop을 구현했다.

Line 41 & 42: Dynamic programming에서는 full backup 방식으로, 모든 state에 접근하여 각 state에서의 value function을 구한다. 따라서, 이를 grid world의 width와 height에 해당하는 for loop들을 만들어서 grid world의 모든 state에 대해 policy evaluation이 가능하도록 구현하였다.

Line 43 & 44: 만약 현재 grid world 좌표의 위치가 obstacle의 위치에 해당된다면, 해당 위치는 agent 가 가질 수 없는 state에 해당되므로 state value 값을 0으로 설정한다.

Line 45: 그 이외의 grid world 좌표들은 agent가 state로 가질 수 있으므로 이에 대해서 full backup을 진행한다.

Line 47: 현재 state에서 action을 취했을 때 얻을 수 있는 future return의 기댓값은 곧 현 state에서의 state value function이 된다. 이때, 각각의 action에 따른 future return의 기댓값을 구하기 위해 action\_sum이라는 길이 4짜리 행렬을 선언하여 0행렬로 초기화하였다. 이 행렬은 각각의 index에 해당하는 ( 현재 state에서 해당 action을 취할 확률 ) x ( 현재 state, action하에서 얻을 수 있는 immediate reward와 future return의 기댓값 )을 계산한 값을 넣어 준다.

Line 48: 현재 state에서 취할 수 있는 모든 action에 대해 ( 현재 state에서 해당 action을 취할 확률 ) x ( 현재 state, action하에서 얻을 수 있는 immediate reward와 future return의 기댓값 )을 계산해 주기 위해 현재 state에서 취할 수 있는 모든 action에 대한 for loop을 돌린다.

Line 50: Dynamic programming에서는 environment에서의 full interaction을 알기 때문에, 현재 state와 action이 주어지면 interaction에 따른 next state와 immediate return을 얻을 수 있다. 이를 env.interaction 모듈을 통해 구현하였다.

Line 52 & 53: ( 현재 state에서 해당 action을 취할 확률 ) x ( 현재 state, action하에서 얻을 수 있는 immediate reward와 future return의 기댓값 ) =  $\text{policy}(a|s) \times (r + \gamma v(s'))$ 을 계산한 값을 action\_sum 행렬에 넣어 준다. 이 식은 Bellman equation에 state transition probability = 1, discount rate ( $\gamma$ ) = 1임을 적용한 식이다.

Line 55: 현재 state에서 취할 수 있는 모든 action들에 대해 ( 현재 state에서 해당 action을 취할 확률 ) x ( 현재 state, action하에서 얻을 수 있는 immediate reward와 future return의 기댓값 )을 모두 더하면 현재 state에서 주어진 policy를 따르는 state value를 구할 수 있다. 따라서, action\_sum에 있는 모든 원소들의 합을 구하여 new\_state\_values 행렬의 현재 state에 해당하는 index에 값을 넣어줌으로써 value function을 update 해준다.

Line 57: 주어진 policy를 따르는 새롭게 얻은 value function이 기존의 value function으로부터 얼마나 변했는지를 확인하기 위해, 각 state 중에서 value function의 변화량 절댓값이 가장 큰 값을 delta로 지정한다. 이를 abs와 max 함수를 통해 구현하였다.

Line 58 & 59: 앞서 설명한 것처럼, delta 값이 threshold 값보다 작을 경우 Policy Evaluation Loop 반복

을 종료한다.

Line 60: Policy evaluation loop의 반복 횟수를 나타내는 iteration flag의 값을 1만큼 증가시켜 준다.

Line 62: 기존 state value function에 해당하는 old\_state\_values를 new\_state\_values 값들로 update한다.

Line 63: 현재 policy iteration에 해당하는 policy evaluation loop의 반복 횟수를 콘솔 창에 print 한다.

## 2.2. “agent.py”의 “Policy Improvement” 부분에 대한 자세한 주석

Line 75: Policy Iteration을 돌릴 때 policy가 더 이상 업데이트되지 않는 경우 iteration을 종료해야 한다. 이를 판별하는 flag 변수인 policy\_stable 값을 True로 초기화하였다.

Line 77 & 78: Dynamic programming에서는 full backup 방식으로 greedy improvement를 진행하는데, 모든 state에 접근하여 각 state에서의 policy improvement를 수행한다. 따라서, 이를 grid world의 width 와 height에 해당하는 for loop들을 만들어서 grid world의 모든 state에 대해 policy improvement가 가능하도록 구현하였다.

Line 80: 현재 state에서의 특정 action을 취했을 때의 future return값 (=action value function 값)을 저장하기 위해 action\_value라는 행렬을 선언하고, 이 행렬을 0행렬로 초기화 해 준다.

Line 82: 현재 state에서 취할 수 있는 각각의 action에 대해 future return 값을 구하기 위해 모든 action에 대한 for loop을 돌린다.

Line 84: Dynamic programming에서는 environment에서의 full interaction을 알기 때문에, 현재 state와 action이 주어지면 interaction에 따른 next state와 immediate return을 얻을 수 있다. 이를 env.interaction 모듈을 통해 구현하였다.

Line 86: (현재 state, action에서 얻을 수 있는 immediate reward와 future return의 기댓값) =  $(r + \gamma v(s')) = q(s, a)$ 을 계산한 값을 action\_value 행렬에 넣어 준다. 이 식은 Bellman equation of action value function에 state transition probability = 1, discount rate ( $\gamma$ ) = 1임을 적용한 식이다.

Line 88: 앞서 계산한 현재 state에서 취한 action들 중 future return (action value function)이 최대로 나타나는 action들을 찾아내기 위해 argmax 연산을 구현했다. 이때, maximum 값이 두 군데 이상에서 나타날 수 있기 때문에 np.argmax가 아닌 np.argwhere 함수를 사용했다. 이렇게 얻게 된 argmax action들은 indices라는 array에 저장된다.

Line 90: 현재 state에서 각 action에 대하여 greedy하게 improve된 policy를 선언해주기 위하여 모든 action에 대한 for loop을 돌린다.

Line 92 & 94: 현재 action이 argmax에 해당되는 경우 (즉 indices array안에 포함되어 있는 경우), 현재 state에 대한 해당 action의 policy를  $1/(argmax\ action\ 들의\ 총\ 개수)$ 로 설정한다.

Line 96 & 98: 현재 action이 argmax에 해당하지 않는 경우(즉 indices array안에 포함되어 있지 않은 경우), 현재 state에 대한 해당 action의 policy를 0으로 설정한다.

Line 100 & 102: Policy의 개선이 있는지의 여부를 판별하여 만약 개선이 있는 경우 policy\_stable이라는 flag 값을 False로 설정해 준다. 이때, policy의 개선 여부 판별은 현재 policy improvement를 수행하기 이전 policy와 이후 policy의 행렬 값을 비교하여, 모든 element가 일치하면 개선이 되지 않는다고 보고, 어떤 element라도 일치하지 않는다면 개선이 되었다고 본다.

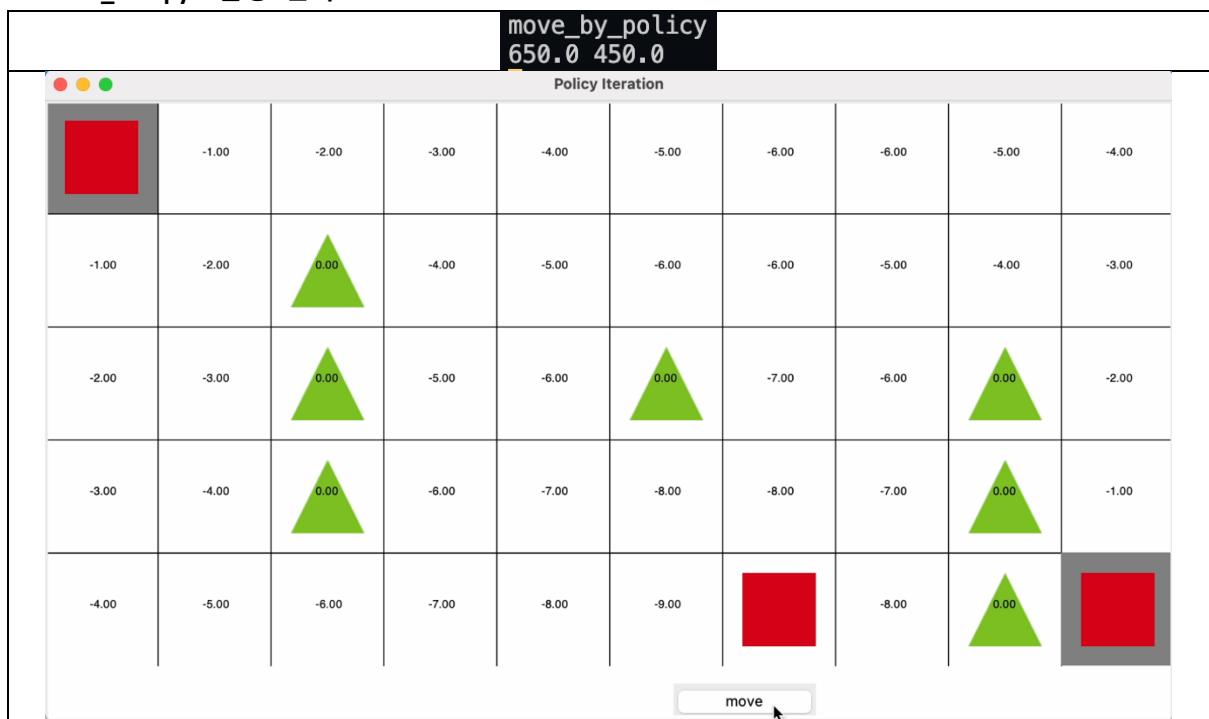
### 3. “main\_train.py” 실행 결과

Policy Evaluation											Policy Improvement										
Policy Evaluation: Iteration:1											Policy Improvement: Iteration:1										
1	-60.41	-109.33	-154.25	-174.66	-183.93	-186.65	-182.3	-171.28	-159.55		1	↔	↔	↔	↔	↔	↔	→	→	→	↓
2	-51.54	-67.9	X	-174.76	-181.81	-186.49	-189.72	-184.96	-168.02	-143.81	2	↑	↔	X	↑	↑	↔	→	→	→	↓
3	-82.71	-87.76	X	-184.22	-187.34	X	-196.81	-195.79	X	-99.87	3	↑	↑	X	↑	↑	X	↑	↑	X	↓
4	-104.83	-108.66	X	-186.57	-191.99	-198.6	-200.91	-201.61	X	-51.94	4	↑	↑	X	↓	↔	↔	↑	↑	X	↓
5	-119.12	-129.41	-156.45	-179.49	-191.46	-198.9	-202.65	-204.12	X		5	↑	↑	↔	↔	↔	↔	↑	↑	X	
	1	2	3	4	5	6	7	8	9	10		1	2	3	4	5	6	7	8	9	10
Policy Evaluation: Iteration:2											Policy Improvement: Iteration:2										
1	-1.0	-2.0	-3.0	-4.0	-5.0	-7.0	-6.0	-5.0	-4.0		1	↔	↔	↔	↔	↔	↔	→↓	→↓	↓	
2	-1.0	-2.0	X	-4.0	-5.0	-6.0	-6.0	-5.0	-4.0	-3.0	2	↑	↔↑	X	↑	↔↑	↔↑	→	→	→	↓
3	-2.0	-3.0	X	-5.0	-6.0	X	-7.0	-6.0	X	-2.0	3	↑	↔↑	X	↑	↔↑	X	→↑	↑	X	↓
4	-3.0	-4.0	X	-8.0	-9.0	-10.0	-8.0	-7.0	X	-1.0	4	↑	↔↑	X	↑	↑	→	→↑	↑	X	↓
5	-4.0	-5.0	-6.0	-7.0	-8.0	-9.0	-10.0	-8.0	X		5	↑	↔↑	↔	↔↑	↔↑	↔	→↑	↑	X	
	1	2	3	4	5	6	7	8	9	10		1	2	3	4	5	6	7	8	9	10
Policy Evaluation: Iteration:3											Policy Improvement: Iteration:3										
1	-1.0	-2.0	-3.0	-4.0	-5.0	-6.0	-6.0	-5.0	-4.0		1	↔	↔	↔	↔	↔	↔	→↓	→↓	↓	
2	-1.0	-2.0	X	-4.0	-5.0	-6.0	-6.0	-5.0	-4.0	-3.0	2	↑	↔↑	X	↑	↔↑	↔↑	→	→	→	↓
3	-2.0	-3.0	X	-5.0	-6.0	X	-7.0	-6.0	X	-2.0	3	↑	↔↑	X	↑	↔↑	X	→↑	↑	X	↓
4	-3.0	-4.0	X	-6.0	-7.0	-9.0	-8.0	-7.0	X	-1.0	4	↑	↔↑	X	↑	↔↑	↔	→↑	↑	X	↓
5	-4.0	-5.0	-6.0	-7.0	-8.0	-9.0	-9.0	-8.0	X		5	↑	↔↑	↔	↔↑	↔↑	↔	→↑	↑	X	
	1	2	3	4	5	6	7	8	9	10		1	2	3	4	5	6	7	8	9	10
Policy Evaluation: Iteration:4											Policy Improvement: Iteration:4										
1	-1.0	-2.0	-3.0	-4.0	-5.0	-6.0	-6.0	-5.0	-4.0		1	↔	↔	↔	↔	↔	↔	→↓	→↓	↓	
2	-1.0	-2.0	X	-4.0	-5.0	-6.0	-6.0	-5.0	-4.0	-3.0	2	↑	↔↑	X	↑	↔↑	↔↑	→	→	→	↓
3	-2.0	-3.0	X	-5.0	-6.0	X	-7.0	-6.0	X	-2.0	3	↑	↔↑	X	↑	↔↑	X	→↑	↑	X	↓
4	-3.0	-4.0	X	-6.0	-7.0	-8.0	-8.0	-7.0	X	-1.0	4	↑	↔↑	X	↑	↔↑	↔	→↑	↑	X	↓
5	-4.0	-5.0	-6.0	-7.0	-8.0	-9.0	-9.0	-8.0	X		5	↑	↔↑	↔	↔↑	↔↑	↔	→↑	↑	X	
	1	2	3	4	5	6	7	8	9	10		1	2	3	4	5	6	7	8	9	10

Policy Evaluation: Iteration:5											Policy Improvement: Iteration:5										
1	-1.0	-2.0	-3.0	-4.0	-5.0	-6.0	-6.0	-5.0	-4.0		1	←	←	←	←	←	←	←	→↓	→↓	↓
2	-1.0	-2.0	X	-4.0	-5.0	-6.0	-6.0	-5.0	-4.0	-3.0	2	↑	↔↑	X	↑	↔↑	↔↑	→	→	→	↓
3	-2.0	-3.0	X	-5.0	-6.0	X	-7.0	-6.0	X	-2.0	3	↑	↔↑	X	↑	↔↑	X	→↑	↑	X	↓
4	-3.0	-4.0	X	-6.0	-7.0	-8.0	-8.0	-7.0	X	-1.0	4	↑	↔↑	X	↑	↔↑	←	→↑	↑	X	↓
5	-4.0	-5.0	-6.0	-7.0	-8.0	-9.0	-9.0	-8.0	X		5	↑	↔↑	←	↔↑	↔↑	↔↑	→↑	↑	X	
1	2	3	4	5	6	7	8	9	10		1	2	3	4	5	6	7	8	9	10	

< 표 1. main\_train.py를 실행하여 Policy Iteration 실행 결과 >

#### 4. "main\_test.py" 실행 결과



Policy Iteration									
	-1.00	-2.00	-3.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00
-1.00	-2.00	 0.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00	-3.00
-2.00	-3.00	 0.00	-5.00	-6.00	 0.00	-7.00	-6.00	 0.00	-2.00
-3.00	-4.00	 0.00	-6.00	-7.00	-8.00	-8.00	-7.00	 0.00	-1.00
-4.00	-5.00	-6.00	-7.00	-8.00	-9.00	-9.00	 0.00	 0.00	

Policy Iteration									
	-1.00	-2.00	-3.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00
-1.00	-2.00	 0.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00	-3.00
-2.00	-3.00	 0.00	-5.00	-6.00	 0.00	-7.00	-6.00	 0.00	-2.00
-3.00	-4.00	 0.00	-6.00	-7.00	-8.00	-8.00	 0.00	 0.00	-1.00
-4.00	-5.00	-6.00	-7.00	-8.00	-9.00	-9.00	-8.00	 0.00	

Policy Iteration									
	-1.00	-2.00	-3.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00
-1.00	-2.00	 0.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00	-3.00
-2.00	-3.00	 0.00	-5.00	-6.00	 0.00	-7.00	 0.00	 0.00	-2.00
-3.00	-4.00	 0.00	-6.00	-7.00	-8.00	-8.00	-7.00	 0.00	-1.00
-4.00	-5.00	-6.00	-7.00	-8.00	-9.00	-9.00	-8.00	 0.00	

Policy Iteration									
	-1.00	-2.00	-3.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00
	-1.00	-2.00	0.00	-4.00	-5.00	-6.00	-6.00	0.00	-3.00
	-2.00	-3.00	0.00	-5.00	-6.00	0.00	-7.00	-6.00	-2.00
	-3.00	-4.00	0.00	-6.00	-7.00	-8.00	-8.00	-7.00	-1.00
	-4.00	-5.00	-6.00	-7.00	-8.00	-9.00	-9.00	0.00	0.00

Policy Iteration									
	-1.00	-2.00	-3.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00
	-1.00	-2.00	0.00	-4.00	-5.00	-6.00	-6.00	0.00	-3.00
	-2.00	-3.00	0.00	-5.00	-6.00	0.00	-7.00	-6.00	0.00
	-3.00	-4.00	0.00	-6.00	-7.00	-8.00	-8.00	-7.00	0.00
	-4.00	-5.00	-6.00	-7.00	-8.00	-9.00	-9.00	-8.00	0.00

Policy Iteration									
	-1.00	-2.00	-3.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00
-1.00	-2.00	 0.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00	
-2.00	-3.00	 0.00	-5.00	-6.00	 0.00	-7.00	-6.00	 0.00	-2.00
-3.00	-4.00	 0.00	-6.00	-7.00	-8.00	-8.00	-7.00	 0.00	-1.00
-4.00	-5.00	-6.00	-7.00	-8.00	-9.00	-9.00	-8.00	 0.00	

Policy Iteration									
	-1.00	-2.00	-3.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00
-1.00	-2.00	 0.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00	-3.00
-2.00	-3.00	 0.00	-5.00	-6.00	 0.00	-7.00	-6.00	 0.00	
-3.00	-4.00	 0.00	-6.00	-7.00	-8.00	-8.00	-7.00	 0.00	-1.00
-4.00	-5.00	-6.00	-7.00	-8.00	-9.00	-9.00	-8.00	 0.00	
	-1.00	-2.00	-3.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00
-1.00	-2.00	 0.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00	-3.00
-2.00	-3.00	 0.00	-5.00	-6.00	 0.00	-7.00	-6.00	 0.00	-2.00
-3.00	-4.00	 0.00	-6.00	-7.00	-8.00	-8.00	-7.00	 0.00	
-4.00	-5.00	-6.00	-7.00	-8.00	-9.00	-9.00	-8.00	 0.00	
	-1.00	-2.00	-3.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00
-1.00	-2.00	 0.00	-4.00	-5.00	-6.00	-6.00	-5.00	-4.00	-3.00
-2.00	-3.00	 0.00	-5.00	-6.00	 0.00	-7.00	-6.00	 0.00	-2.00
-3.00	-4.00	 0.00	-6.00	-7.00	-8.00	-8.00	-7.00	 0.00	-1.00
-4.00	-5.00	-6.00	-7.00	-8.00	-9.00	-9.00	-8.00	 0.00	

< 표 2. main\_test.py를 실행하여 move 버튼 클릭 후 goal 까지의 과정 기록 >

## 5. 결과에 대한 분석

```
policy iteration #: 1, policy evaluation iteration #: 1216
policy stable False:
policy iteration #: 2, policy evaluation iteration #: 10
policy stable False:
policy iteration #: 3, policy evaluation iteration #: 1
policy stable False:
policy iteration #: 4, policy evaluation iteration #: 1
policy stable False:
policy iteration #: 5, policy evaluation iteration #: 0
policy stable True:
```

< 그림 3. Policy iteration 횟수와 그에 따른 policy evaluation loop 반복 횟수 및 policy stable 여부 >

- 표 1과 그림 3에서 확인할 수 있듯이, 총 5번의 policy iteration을 통해 optimal policy와 optimal value function을 얻을 수 있었다.
- 첫 policy evaluation은 equiprobable policy로 초기화 된 policy 하에서의 state value function을 구한 것으로, 각각의 state value들의 값이 -50부터 -200으로 매우 큰 절댓값을 보인다. 또한, state의 거리가 goal에서부터 멀수록 state value는 급격히 증가하는 것을 볼 수 있다. 이는 agent가 무작위로 action을 선택할 경우 장애물 혹은 boarder를 피하면서 goal에 도달하는 데 필요한 이동 횟수가 매우 많음을 의미한다.
- 그림 3에서 policy iteration의 횟수가 증가할수록, policy evaluation의 반복 횟수는 기하급수적으로 줄어듦을 확인할 수 있다. 첫 번째 policy iteration에서 evaluation은 1216번 반복된 반면, 두 번째에서는 10번, 그리고 마지막에서는 반복이 아예 되지 않았다. 그 이유는 두 번째 이후 evaluation시 old value function의 초기화를 직전에 구한 value function으로 해주었기 때문에 수렴이 더 빨라졌다 고 볼 수 있다.
- 표 1을 통해 첫 번째 iteration에서 policy improvement 이후 얻어진 policy는 즉각적으로 상당히 개선된 결과를 나타냄을 알 수 있으며, 그 값은 optimal policy와 이미 매우 근접했음을 확인할 수 있다. 그 이유는 주어진 task가 매우 간단해서, 처음 계산한 state value들로도 매우 유의미한 policy improvement가 가능했음을 보여 준다.
- 표 1에서 두 번째부터 마지막까지의 policy evaluation과 policy improvement 시 얻게 되는 state value들과 policy들은 사소한 개선이 이뤄짐을 알 수 있다. 이는 이미 두 번째 iteration부터 state value들과 policy들이 optimal value function과 optimal policy에 거의 근접했음을 의미한다.
- 특히, 표 1에서 4번째와 5번째 policy evaluation과 policy improvement를 통해 얻게 되는 state value들과 policy들이 완전히 동일함을 발견할 수 있다. 이는 이미 4번째 policy iteration에서 state value들과 policy가 optimal한 값으로 수렴했음을 의미한다. 5번째 iteration을 통해 이전에 구한 policy와 해당 iteration에서 구한 policy가 동일함을 확인했고, 따라서 policy iteration 반복이 종료된 것이다.
- 표 1을 통해 policy improvement는 greedy한 policy improvement 전략을 따르는 것을 확인할 수 있다. 이는 evaluation을 통해 구한 state value들 중에서 다음 이동 시 가장 state value 값이 큰 방향으로 움직이는 policy를 정하고 있는 것을 통해 확인할 수 있다. 다음 이동 시 state value 값이 가장 큰 방향이 여러 곳이면, policy는 각 해당 방향으로 진행하도록 improve 되며, 이때의 policy는 해당 action들에 대해 stochastic하게 정의된다. 이는 표 1에서 화살표가 2개 이상인 policy들로 나타난다.
- 표 2를 통해 agent가 장애물을 피해 가장 빠른 경로로 goal 지점에 도달하는 action을 가져가는 것을 확인할 수 있었다.

- 또한, 표 2에서 grid에 표시된 값들은 해당 state에서의 optimal state value들로, 여기서의 다음 action은 greedy하게, 즉 next state들 중에서 optimal state value가 가장 큰 state로 이동하는 policy를 가져가게 된다. 이 경우에는 해당 policy가 곧 optimal policy이다.
- 최종적으로 얻게 된 policy가 optimal policy이므로, 우리가 reward를 탈출했을 때 0, 그 외의 경우 -1로 정의할 때 의도했던 것처럼 agent는 최단 경로를 통해 grid world를 가능한 빨리 탈출하는 행동을 취한다. 이때 optimal state value의 절댓값은 goal로부터 현재 state까지의 최단 탈출 경로의 길이와 같다. 이는 표 1, 표 2를 통해 확인할 수 있었다.