< 지능제어 3차 숙제 결과보고서 >

2018142102 홍의진

1. “agent.py”의 “write your code here” 부분 code

1.1. “agent.py” Monte_Carlo_Control 함수의 Sequence Generation 부분 code

```
70      # 이전 time step에서 해당 state를 방문했는지를 기록하는 array를 생성
71      visited = np.zeros((HEIGHT, WIDTH))
72
73      # Sequence generation
74      # done, timeout이라는 flag 값들을 이용하여 while문 반복을 제어
75      while (done == False) and (timeout == False):
76          # get_action 함수를 통해 policy에 따른 현재 state의 action을 불러옴
77          action_index = self.get_action(state)
78          # action_index에 해당하는 action을 action이란 변수로 지정
79          action = ACTIONS[action_index]
80          # env 객체의 interaction method를 이용하여 현재 state, action에 대한 next_state, reward를 구함
81          next_state, reward = self.env.interaction(state, action)
82          # history list에 (s, a, s', r)를 저장한다
83          history.append((state, action_index, next_state, reward))
84          # 현재 방문한 state를 visited array의 해당 state 자리에 count를 1만큼 늘려줌
85          visited[state[0], state[1]] += 1
86
87          # done이 true이면 agent가 terminal state에 도달했다는 것을 의미
88          if (next_state in self.env.goal):
89              done = True
90
91          # timeout이 true이면 generated sequence의 길이가 max_seq_len에 도달했다는 것을 의미
92          if (seq_len == max_seq_len):
93              timeout = True
94          # timeout이 발생하지 않았을 경우
95          else:
96              # seq_len을 1만큼 늘려주고
97              seq_len += 1
98              # 현재 state를 next_state값으로 update 함
99              state = next_state
100
101     # timeout이 발생한 경우엔 생성한 episode를 policy update에 사용하지 않기 위해 조건문을 지정
102     if timeout == False:
```

<그림1. “agent.py” Monte_Carlo_Control 함수의 Sequence Generation 부분 code>

1.2. “agent.py” Monte_Carlo_Control 함수의 Q value and policy update 부분 code

```
        # cum_reward는 현재 sequence에서 누적된 reward값을 의미
        cum_reward = 0
        # Q Value and policy update
        # sequence의 길이만큼 for문을 반복한다.
        for index in range(seq_len):
            # history list에 저장되어 있는 (s, a, s', r) pair를 불러옴
            ([i, j], a, _, r) = history[(seq_len-1) - index]
            # 누적된 reward에 discount rate을 적용하여 현재 time step에서의 reward도 더해줌
            cum_reward = discount * cum_reward + r
            # first-visit MC를 적용
            # 만약 현재 time step에서 해당 state를 방문한 게 처음이 아니면
            if (visited[i, j] > 1):
                # visited 행렬에 저장되어 있는 값에서 1만큼을 빼줌
                visited[i, j] -= 1
            # 만약 현재 time step에서 해당 state를 방문한 게 처음이라면
            else:
                # 해당 state, action에 해당하는 action value function(Q)을 constant alpha MC로 update
                self.Q_values[i][j][a] += alpha * (cum_reward - self.Q_values[i][j][a])
                # epsilon-greedy하게 policy를 update하기 위해 Q값이 최대가 되게 하는 action index들을 구함
                indices = np.argmax(self.Q_values[i][j])
                # policy를 epsilon-greedy하게 update를 시켜주기 위해 각 action에 접근하도록 for loop을 돌림
                for k, actions in enumerate(self.ACTIONS):
                    # 현재 action index가 greedy하게 찾은 action index에 해당된다면
                    if (k == indices):
                        # 현재 state, action의 policy를 epsilon-greedy하게 update 해줌
                        self.policy[i, j, k] = (((1 - epsilon)) + (epsilon / len(self.ACTIONS)))
                    # 현재 action index가 greedy하게 찾은 action index에 해당되지 않는다면
                    else:
                        # 현재 state, action의 policy를 epsilon-greedy하게 update 해줌
                        self.policy[i, j, k] = epsilon / len(self.ACTIONS)
            # computation이 끝나는 시점의 시간을 기록
            end = time.time()
            # 20000번씩의 episode마다 경과 시간의 평균값을 구하기 위해 각각의 경과시간을 합함
            avg_time += (end - start) * 1000 / 20000
            # 현재 종료 시점의 시간을 다음 시작 시점의 시간으로 설정
            start = end
        # 매번 decay_period의 배수만큼 episode에 대한 계산이 진행될 때마다
        if (episode % decay_period == 0) and (episode != 0):
            # optimal policy로 수렴시키기 위해 epsilon을 decay_rate만큼 곱하여 감소
            epsilon *= decay_rate
            # 현재 episode가 끝난 시점에 대한 시간 값을 저장
            end = time.time()
            # decay_period의 배수만큼 반복될때마다 Q function의 평균값을 저장
            Q_avg.append(np.average(self.Q_values))
            # decay_period의 배수만큼 반복될때마다 현재 episode에 대한 연산에서 경과된 시간의 평균을 저장
            elapsed_time.append(avg_time)
            avg_time = 0  # 20000번째 episode마다 average time을 0으로 초기화
```

<그림2. “agent.py” Monte_Carlo_Control 함수의 Q value and policy update 부분 code>

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

2.1. “agent.py” Monte_Carlo_Control 함수의 Sequence Generation 부분에 대한 자세한 주석

Line 71: 이전 time step에서 해당 state를 방문했는지를 기록하는 array를 생성해 주었다.

Line 75: done, timeout이라는 flag 값들을 이용하여 while문 반복을 제어한다.

Line 77: get_action 함수를 통해 policy에 따른 현재 state의 action을 불러온다.

Line 79: action_index에 해당하는 action을 action이란 변수로 지정한다.

Line 81: env 객체의 interaction method를 이용하여 현재 state, action에 대한 next_state, reward를 구하였다.

Line 83: history list에 (s, a, s', r)를 저장한다.
Line 85: 현재 방문한 state를 visited array의 해당 state 자리에 count를 1만큼 늘려준다.
Line 88 & 89: done이 true이면 agent가 terminal state에 도달했다는 것을 의미한다.
Line 92 & 93: timeout이 true이면 generated sequence의 길이가 max_seq_len에 도달했다는 것을 의미한다.
Line 95 & 97 & 99: timeout이 발생하지 않았을 경우 seq_len을 1만큼 늘려주고 현재 state를 next_state값으로 update 해준다.
Line 102: timeout이 발생한 경우엔 생성한 episode를 policy update에 사용하지 않기 위해 조건문을 지정해 주었다.

2.2. "agent.py" Monte_Carlo_Control 함수의 Q value and policy update 부분에 대한 자세한 주석

Line 105: cum_reward는 현재 sequence에서 누적된 reward값을 의미한다.
Line 108: sequence의 길이만큼 for문을 반복한다.
Line 110: history list에 저장되어 있는 (s, a, s', r) pair를 불러온다.
Line 112: 누적된 reward에 discount rate을 적용하여 현재 time step에서의 reward도 더해준다.
Line 115 & 117: First-visit MC를 적용하여 만약 현재 time step에서 해당 state를 방문한 게 처음이 아니면 visited 행렬에 저장되어 있는 값에서 1만큼을 빼준다.
Line 119 & 121: 만약 현재 time step에서 해당 state를 방문한 게 처음이라면 해당 state, action에 해당하는 action value function(Q)을 constant alpha MC로 update 해준다.
Line 123: epsilon-greedy하게 policy를 update하기 위해 Q값이 최대가 되게 하는 action index들을 구하였다.
Line 125: policy를 epsilon-greedy하게 update를 시켜주기 위해 각 action에 접근하도록 for loop을 돌렸다.
Line 127 & 129: 현재 action index가 greedy하게 찾은 action index에 해당된다면 현재 state, action의 policy를 epsilon-greedy하게 update 해준다.
Line 131 & 133: 현재 action index가 greedy하게 찾은 action index에 해당되지 않는다면 현재 state, action의 policy를 epsilon-greedy하게 update 해준다.
Line 135: computation이 끝나는 시점의 시간을 기록한다.
Line 137: 20000번씩의 episode마다 경과 시간의 평균값을 구하기 위해 각각의 경과시간을 합한다.
Line 139: 현재 종료 시점의 시간을 다음 시작 시점의 시간으로 설정한다.
Line 141 & 143: 매번 decay_period의 배수만큼 episode에 대한 계산이 진행될 때마다 optimal policy로 수렴시키기 위해 epsilon을 decay_rate만큼 곱하여 감소시킨다.
Line 145: 현재 episode가 끝난 시점에 대한 시간 값을 저장한다.
Line 147: decay_period의 배수만큼 반복될때마다 Q function의 평균값을 저장한다.
Line 149: decay_period의 배수만큼 반복될때마다 현재 episode에 대한 연산에서 경과된 시간의 평균을 저장한다.
Line 150: 20000번째 episode마다 average time을 0으로 초기화한다.

3. “main_train.py” code 실행 결과

Policy Evaluation: Iteration:1

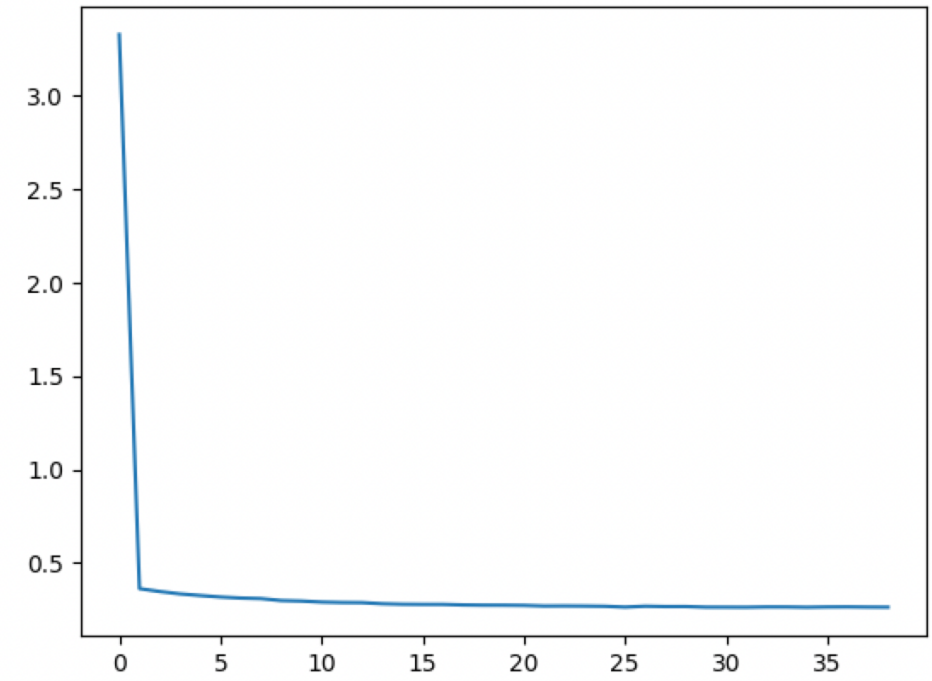
1	-17.07	-16.11	X	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
2	-16.09	-15.09	X	-9.04	-8.03	-7.02	-6.01	-5.01	-4.01	-3.0
3	-15.08	-14.11	X	-10.06	X	-8.03	X	-4.01	-3.01	-2.0
4	-14.09	-13.09	-12.08	-11.05	X	-9.05	X	-3.01	-2.01	-1.0
5	-15.15	-14.13	-13.1	-12.07	-11.06	-10.06	X	-2.01	-1.0	
	1	2	3	4	5	6	7	8	9	10

<그림3. Monte Carlo Control에서 구한 state value function의 값>

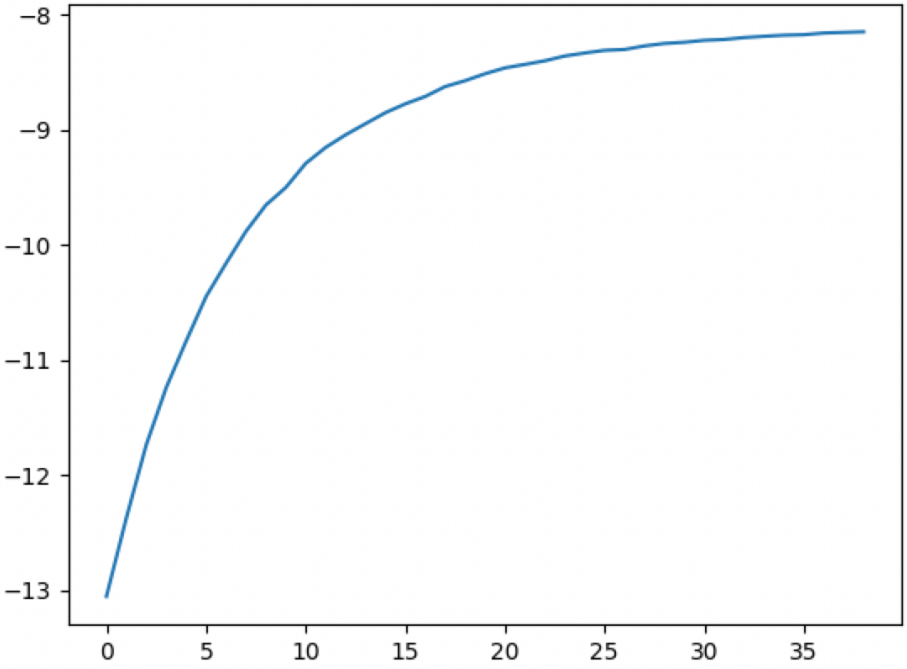
Policy Improvement: Iteration:1

1	↓	↓	X	→	→	↓	↓	↓	→	↓
2	↓	↓	X	→	→	→	→	↓	→	↓
3	↓	↓	X	↑	X	↑	X	↓	→	↓
4	→	→	→	↑	X	↑	X	→	→	↓
5	→	→	→	→	→	↑	X	→	→	
	1	2	3	4	5	6	7	8	9	10

<그림4. Monte Carlo Control에서 구한 policy의 값>

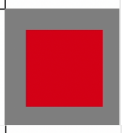
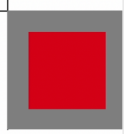
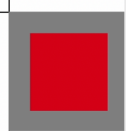


<그림5. Iteration의 진행에 따른 average computational time 그래프>



<그림6. Iteration에 따른 average Q value 그래프>

4. “main_test.py” code 실행 결과

	-16.11	 0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	 0.00	-9.04	-8.03	-7.02	-6.01	-5.01	-4.01	-3.00
-15.08	-14.11	 0.00	-10.06	 0.00	-8.03	 0.00	-4.01	-3.01	-2.00
-14.09	-13.09	-12.08	-11.05	 0.00	-9.05	 0.00	-3.01	-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	 0.00	-2.01	-1.00	
-17.07	-16.11	 0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
	-15.09	 0.00	-9.04	-8.03	-7.02	-6.01	-5.01	-4.01	-3.00
-15.08	-14.11	 0.00	-10.06	 0.00	-8.03	 0.00	-4.01	-3.01	-2.00
-14.09	-13.09	-12.08	-11.05	 0.00	-9.05	 0.00	-3.01	-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	 0.00	-2.01	-1.00	
-17.07	-16.11	 0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	 0.00	-9.04	-8.03	-7.02	-6.01	-5.01	-4.01	-3.00
	-14.11	 0.00	-10.06	 0.00	-8.03	 0.00	-4.01	-3.01	-2.00
-14.09	-13.09	-12.08	-11.05	 0.00	-9.05	 0.00	-3.01	-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	 0.00	-2.01	-1.00	

-17.07	-16.11	0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	0.00	-9.04	-8.03	-7.02	-6.01	-5.01	-4.01	-3.00
-15.08	-14.11	0.00	-10.06	0.00	-8.03	0.00	-4.01	-3.01	-2.00
	-13.09	-12.08	-11.05	0.00	-9.05	0.00	-3.01	-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	0.00	-2.01	-1.00	
-17.07	-16.11	0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	0.00	-9.04	-8.03	-7.02	-6.01	-5.01	-4.01	-3.00
-15.08	-14.11	0.00	-10.06	0.00	-8.03	0.00	-4.01	-3.01	-2.00
-14.09		-12.08	-11.05	0.00	-9.05	0.00	-3.01	-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	0.00	-2.01	-1.00	
-17.07	-16.11	0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	0.00	-9.04	-8.03	-7.02	-6.01	-5.01	-4.01	-3.00
-15.08	-14.11	0.00	-10.06	0.00	-8.03	0.00	-4.01	-3.01	-2.00
-14.09	-13.09		-11.05	0.00	-9.05	0.00	-3.01	-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	0.00	-2.01	-1.00	

-17.07	-16.11	0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	0.00	-9.04	-8.03	-7.02	-6.01	-5.01	-4.01	-3.00
-15.08	-14.11	0.00	-10.06	0.00	-8.03	0.00	-4.01	-3.01	-2.00
-14.09	-13.09	-12.08		0.00	-9.05	0.00	-3.01	-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	0.00	-2.01	-1.00	
-17.07	-16.11	0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	0.00	-9.04	-8.03	-7.02	-6.01	-5.01	-4.01	-3.00
-15.08	-14.11	0.00		0.00	-8.03	0.00	-4.01	-3.01	-2.00
-14.09	-13.09	-12.08	-11.05	0.00	-9.05	0.00	-3.01	-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	0.00	-2.01	-1.00	
-17.07	-16.11	0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	0.00		-8.03	-7.02	-6.01	-5.01	-4.01	-3.00
-15.08	-14.11	0.00	-10.06	0.00	-8.03	0.00	-4.01	-3.01	-2.00
-14.09	-13.09	-12.08	-11.05	0.00	-9.05	0.00	-3.01	-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	0.00	-2.01	-1.00	

-17.07	-16.11	0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	0.00	-9.04		-7.02	-6.01	-5.01	-4.01	-3.00
-15.08	-14.11	0.00	-10.06	0.00	-8.03	0.00	-4.01	-3.01	-2.00
-14.09	-13.09	-12.08	-11.05	0.00	-9.05	0.00	-3.01	-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	0.00	-2.01	-1.00	

-17.07	-16.11	0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	0.00	-9.04	-8.03		-6.01	-5.01	-4.01	-3.00
-15.08	-14.11	0.00	-10.06	0.00	-8.03	0.00	-4.01	-3.01	-2.00
-14.09	-13.09	-12.08	-11.05	0.00	-9.05	0.00	-3.01	-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	0.00	-2.01	-1.00	

-17.07	-16.11	0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	0.00	-9.04	-8.03	-7.02		-5.01	-4.01	-3.00
-15.08	-14.11	0.00	-10.06	0.00	-8.03	0.00	-4.01	-3.01	-2.00
-14.09	-13.09	-12.08	-11.05	0.00	-9.05	0.00	-3.01	-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	0.00	-2.01	-1.00	

-17.07	-16.11	0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	0.00	-9.04	-8.03	-7.02	-6.01		-4.01	-3.00
-15.08	-14.11	0.00	-10.06	0.00	-8.03	0.00	-4.01	-3.01	-2.00
-14.09	-13.09	-12.08	-11.05	0.00	-9.05	0.00	-3.01	-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	0.00	-2.01	-1.00	
-17.07	-16.11	0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	0.00	-9.04	-8.03	-7.02	-6.01	-5.01	-4.01	-3.00
-15.08	-14.11	0.00	-10.06	0.00	-8.03	0.00		-3.01	-2.00
-14.09	-13.09	-12.08	-11.05	0.00	-9.05	0.00	-3.01	-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	0.00	-2.01	-1.00	
-17.07	-16.11	0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	0.00	-9.04	-8.03	-7.02	-6.01	-5.01	-4.01	-3.00
-15.08	-14.11	0.00	-10.06	0.00	-8.03	0.00	-4.01	-3.01	-2.00
-14.09	-13.09	-12.08	-11.05	0.00	-9.05	0.00		-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	0.00	-2.01	-1.00	

-17.07	-16.11	0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	0.00	-9.04	-8.03	-7.02	-6.01	-5.01	-4.01	-3.00
-15.08	-14.11	0.00	-10.06	0.00	-8.03	0.00	-4.01	-3.01	-2.00
-14.09	-13.09	-12.08	-11.05	0.00	-9.05	0.00	-3.01		-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	0.00	-2.01	-1.00	
-17.07	-16.11	0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	0.00	-9.04	-8.03	-7.02	-6.01	-5.01	-4.01	-3.00
-15.08	-14.11	0.00	-10.06	0.00	-8.03	0.00	-4.01	-3.01	-2.00
-14.09	-13.09	-12.08	-11.05	0.00	-9.05	0.00	-3.01	-2.01	
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	0.00	-2.01	-1.00	
-17.07	-16.11	0.00	-10.03	-9.04	-8.03	-7.07	-6.03	-5.02	-4.01
-16.09	-15.09	0.00	-9.04	-8.03	-7.02	-6.01	-5.01	-4.01	-3.00
-15.08	-14.11	0.00	-10.06	0.00	-8.03	0.00	-4.01	-3.01	-2.00
-14.09	-13.09	-12.08	-11.05	0.00	-9.05	0.00	-3.01	-2.01	-1.00
-15.15	-14.13	-13.10	-12.07	-11.06	-10.06	0.00	-2.01	-1.00	

5. 결과에 대한 분석 (Discussion)

- 그림3을 통해 Monte Carlo Control (MCC)로 구한 state value function이 expected reward값과 거의 유사함을 확인할 수 있었다. 이를 통해 MCC에서 구한 state value function이 optimal value function에 수렴하고 있다는 사실을 알 수 있다. 이는 epsilon-greedy method에서 epsilon의 값이 감소하여 마지막에는 0으로 수렴하기 때문에 state value function이 optimal value function에 수렴한 것이다. 소수점 아래의 값들은 state마다 크기가 다르게 나타났는데, 이는 episode들이 random하게 생성되고 value function update에 반영되기 때문에 상대적으로 value function update가 덜 반영된 state들이 나타난 것이다. 이러한 측면에서 봤을 때, optimal value function에서는 state value가 같게 나와야 하는 state들에서 state value들이 다르게 나타났다. 이는 이후 policy를 구하는 과정에 영향을 준다. 한편, constant-alpha MC를 사용하였기 때문에 policy가 update됨에 따라 action value function update를 통한 policy update에서 새로 생성된 policy를 바탕으로 만들어진 episode이 policy update에 반영되는 가중치가 더 높았다. 그렇기 때문에 update되는 action value function과 policy는 더 빠르게 optimal한 값으로 수렴할 수 있었던 것이다.
- 그림4를 통해 Monte Carlo Control로 구한 policy는 optimal policy와는 다르다는 것을 알 수 있었다. 이는 원래 optimal value function에서 state value가 같게 나와야 하는 state들에서 state value들이 소수점 아래 단위에서 다르게 나타났기 때문에, 특정 state에서의 optimal policy가 두 개 이상의 action으로 이뤄져 있더라도 MCC를 통해 구한 policy는 소수점 아래까지 고려했을 때 state value가 더 큰 state로 이동하는 policy가 된 것이다. 그러나 MCC를 통해 구한 policy 역시 장애물을 피해 최단 경로로 terminal state에 도달하는 이동을 보장하는 정책이므로, 주어진 task의 solution으로는 손색이 없다.
- 그림5를 통해 MCC의 iteration에 따라 생성되는 episode들에 대한 computing time이 급격히 줄어든다는 것을 알 수 있다. 이는 MCC의 value function update를 통해 이뤄지는 policy update로 인해 거듭 생성되는 episode들은 이전보다 optimal policy에 가까워진 policy를 따라 생성되기 때문에 episode의 길이가 더 짧아지고, 이에 따라 computing time역시 감소한다. Episode의 길이가 길수록 episode를 생성하는 시간과 생성된 episode에 대하여 policy와 action value function 값을 update해 주는 과정에서 시간이 더 오래 걸리게 된다.
- 그림 6을 통해 action value function (Q values)의 평균값은 iteration을 거듭함에 따라 그 절댓값이 감소하는 쪽으로 수렴함을 확인할 수 있었다. 이는 Q values update를 통해 state values가 optimal한 값으로 수렴하고 있음을 보여준다.
- main_test.py 코드 실행 결과를 통해 agent가 MCC를 통해 구한 policy를 따라 정해진 task를 잘 수행하고 있음을 확인할 수 있었다. 이때 policy는 앞서 말한 것처럼 optimal한 policy는 아니지만, 주어진 task 수행에 부합하는 policy이다.