AI_Assignment 06 Report

1분반 20191244 김현승

1. compute_transition_matrix

MxNx4xMxN 크기의 numpy 배열 P를 초기화한 후 P[r, c, a, r′, c′]을 계산하기 위해 5개의 for 문에 진입한다.

```
if model.T[r, c] :
    P[r, c, a, r_prime, c_prime] = 0
```

현재 state가 terminal state인 경우 모든 a, r′, c′에 대해 P[r, c, a, r′, c′] = 0이다. 현재 state가 terminal state가 아닌 경우 다음과 같이 확률을 계산했다.

```
if a == 0 : # left(0)
    r_want, c_want = r, max(c-1, 0)
    r_cntclk, c_cntclk = min(r+1, M-1), c
    r_clk, c_clk = max(r-1, 0), c
elif a == 1 : # up(1)
    r_want, c_want = max(r-1, 0), c
    r_cntclk, c_cntclk = r, max(c-1, 0)
    r_clk, c_clk = r, min(c+1, N-1)
elif a == 2 : # right(2)
    r_want, c_want = r, min(c+1, N-1)
    r_cntclk, c_cntclk = max(r-1, 0), c
    r_clk, c_clk = min(r+1, M-1), c
elif a == 3 : # down(3)
    r_want, c_want = min(r+1, M-1), c
    r_cntclk, c_cntclk = r, min(c+1, N-1)
    r_clk, c_clk = r, max(c-1, 0)
```

우선 현재 취하고자 하는 action에 따라 want, cntclk, clk 좌표를 계산했다. want는 현재 state에서 원하는 action을 취했을 때의 state이고, cntclk는 원하는 방향의 반시계 방향으로 action을 취했을 때 state, clk는 원하는 방향의 시계 방향으로 action을 취했을 때의 state이다. 위와 같이 좌표를 구한 다음 want, cntclk, clk의 좌표에 벽이 있는지 확인한다. 벽이 있다면 해당 좌표를 현재 state로 돌려 놓는다.

```
if model.W[r_want, c_want] :
    r_want, c_want = r, c
if model.W[r_cntclk, c_cntclk] :
    r_cntclk, c_cntclk = r, c
if model.W[r_clk, c_clk] :
    r_clk, c_clk = r, c

# r_prime, c_prime이 want, cntclk, clk인 경우 각각 D[r
if r_want == r_prime and c_want == c_prime :
    P[r, c, a, r_prime, c_prime] += model.D[r, c, 0]
if r_cntclk == r_prime and c_cntclk == c_prime :
    P[r, c, a, r_prime, c_prime] += model.D[r, c, 1]
if r_clk == r_prime and c_clk == c_prime :
    P[r, c, a, r_prime, c_prime] += model.D[r, c, 2]
```

이후 want, cntclk, clk의 좌표가 (r′, c′)인 경우 각각 D[r, c, 0], D[r, c, 1], D[r, c, 2] 의 확률을 더해주면 P table의 확률 계산이 완료된다.

2. update_utility

U_next를 MxN numpy 영행렬로 초기화한다. r, c, a에 대해 반복문을 도는데, 이 때 모든 다음 state s', 즉 (r', c')에 대해 P[r, c, a, r', c'] * U_current[r', c']의 합을 구하고, 값을 저장해가며 모든 action에 대해 P[r, c, a, r', c'] * U_current[r', c']의 합의 최댓값을 구한다.

```
for r in range(M) :
    for c in range(N) :
        # U_(i+1)(s) = R(s) + ~~~
        U_next[r, c] += model.R[r, c]
        max_sumU = -np.inf
        for a in range(4) :
            sumU = 0
            for r_prime in range(M) :
                for c_prime in range(N) :
                    # sum of (P[s'|s, a] * U_i(s')) for all s'.
                    sumU += P[r, c, a, r_prime, c_prime] * U_current[r_prime, c_prime]
            # 모든 action에 대해 가장 큰 sumU를 구한다.
            max_sumU = max(max_sumU, sumU)
        # U_(i+1)(s) = R(s) + gamma * max_sumU
        U_next[r, c] += model.gamma * max_sumU
```

최댓값을 구한 후 정해진 감마 값을 더하고, R[r, c]을 더하면 U_next[r, c] 값을 구할 수 있다.

실행 결과

-0.040	-0.040	-0.040	1.000
-0.040	x	-0.040	-1.000
-0.040	-0.040	-0.040	-0.040

3. value_iteration

value_iteration() 함수에서는 update_utility를 반복하며 모든 state에 대해 현재 효용과 다음 효용의 차이가 충분히 작아질 때까지 반복한다.

```
for i in range(100) :
    # update utility
    U_next = update_utility(model, P, U_current)
    # 모든 state에 대해 U_next와 U_current의 차이가 epsi
    result = calculate_delta(model, U_current, U_next)
    if result : break
    # next를 current로 넘기고 다시 반복
    U_current = U_next.copy()
return U_next
```

2번에서 구현한 update_utility() 함수를 이용해 다음 효용을 구하고 calculate_delta 함수로 현재 효용과 다음 효용의 차이를 확인한다. 만약 모든 state에 대해 두 효용의 차가 epsilon보다 작아지면 calculate-delta() 함수는 True를 반환해 위의 for문을 탈출하게 된다. 그렇지 않으면 최대 100

번까지 효용 업데이트를 계속하게 된다.

```
def calculate_delta(model, U_current, U_next) :
    M, N = model.M, model.N
    for r in range(M) :
        for c in range(N) :
            # epsilon보다 커지는 state 있으면 False return.
            if abs(U_current[r, c] - U_next[r, c]) > epsilon :
                return False
    return True
```

위에서 설명했듯, calculate_delta() 함수는 모든 state (r, c)에 대해 U_current와 U_next 값의 차이를 계산하고 epsilon보다 크거나 같은 값을 갖는 state가 있으면 False, 모두 작으면 True를 리턴하는 함수이다. 차이가 epsilon보다 큰 state를 찾은 경우 바로 반복문을 멈추고 return해 조금 더 빠른 실행 시간을 가질 수 있다.

실행 결과

0.812	0.868	0.918	1.000
0.762	x	0.660	-1.000
0.705	0.655	0.611	0.387