

인공지능

Project3

담당교수 : 박철수 교수님

학 번 : 2021202058

성 명 : 송채영

1. Introduction

이번 프로젝트에서는 7x7 크기의 grid world 에서 Dynamic Programming (DP)를 활용하여 주어진 조건을 만족하도록 구현하는 것이 목표이다. 주어진 환경에서 Policy Iteration 과 Value Iteration 을 사용하여 policy 를 최적화해본다. Grid world 는 시작점과 끝점이 있으며 상, 하, 좌, 우 총 4 가지 action 이 가능하다. 더불어 가장 바깥쪽 state 에서는 grid world 밖으로 나가는 action 을 선택하는 경우 제자리로 돌아가도록 한다.

2. Algorithm

우선 프로젝트에서 사용한 알고리즘 및 method 에 대해 설명하겠다.

- Dynamic Programming

Dynamic Programming, DP 는 큰 Programming 을 여러 process 로 나누어 해결한다는 의미이다. 즉 시간에 따라 변하는 상황에서 효과적으로 계획을 세우기 위해 여러 개의 프로세스로 작업을 세분화하는 것을 말한다. 큰 작업을 작은 단위로 나누어 각각의 작은 프로세스가 시간에 따라 변화하는 특성에 대응할 수 있도록 한다.

- Policy evaluation - random Policy

Policy evaluation, 정책 평가는 Reinforcement Learning(강화 학습)에서 사용되는 개념 중 하나로 현재 정책(policy)가 얼마나 좋은지에 대해 측정하는 과정을 의미한다. 여기서 policy 는 agent 가 특정 state 에서 어떤 action 을 선택하는 규칙을 말한다. Random policy 는 agent 가 상태에 상관없이 가능한 행동들을 무작위로 선택하는 policy 를 말한다. 즉 random policy 에서는 agent 가 어떤 상황이든, 가능한 행동들 중 무작위로 하나를 선택한다. 이때 해당 policy 가 얼마나 좋은 성능을 보이는지 측정할 수 있는 state-value-function 을 사용하는데 $V(s) = \mathbb{E}_{\pi}[G_t | S_t = s]$ 수식은 다음과 같다. 이때 $V(s)$ 는 상태 s 의 가치를, \mathbb{E}_{π} 는 정책 π 에 대한 기대 값, G_t 는 시점 t 에서 얻는 반환 값, S_t 는 시점 t 에서의 상태를 나타낸다.

- Policy Improvement

Policy Improvement, 정책 개선은 Reinforcement Learning(강화 학습)에서 사용되는 개념 중 하나로 주어진 환경에서의 agent 의 성능을 향상시키기 위해 현재의 policy 를 수정하는 과정을 의미한다. 주로 Policy Iteration 이나 Value Iteration 과 함께 사용되며 최적의 policy 를 찾는데 활용된다.

- Policy Iteration

Policy Iteration 은 Reinforcement Learning(강화 학습)에서 최적의 policy 를 찾기 위한 반복적인 알고리즘 중 하나로, policy evaluation 과 policy improvement 를 번갈아 수행하며 최적의 policy 를 찾아낸다.

- Value Iteration

Value Iteration 은 Reinforcement Learning(강화 학습)에서 최적의 가치 함수를 찾는 알고리즘 중 하나이다. 이 알고리즘은 policy iteration 과 유사하지만 policy 를 명시적으로 개선하지 않고 value 함수를 반복적으로 업데이트 하여 최적의 value 함수를 찾아낸다. $V(s) \leftarrow \max_a (R(s, a) + \gamma \sum_{s'} P(s' | s, a) V(s'))$ value 함수 업데이트 하는 수식으로 $V(s)$ 는 상태 s 의 가치를, $R(s, a)$ 는 상태 s 에서 행동 a 를 취했을 때 받는 보상을, $P(s' | s, a)$ 는 상태 s 에서 행동 a 를 취한 후 다음 상태가 s' 이 될 확률을, γ 은 discount factor 로 미래의 보상을 현재 가치로 얼마나 강조할지를 결정한다.

- Greedy policy improvement

Greedy(탐욕적) 정책 개선은 Reinforcement Learning(강화 학습)에서 policy 를 개선하는 방법 중 하나로, 현재 가치 함수나 가치 행렬을 기반으로 특정 상태에서 가장 높은 가치를 갖는 행동을 선택하여 policy 를 업데이트 하는 방식을 말한다.

- Bellman Optimality Eqn.

위 방정식은 Reinforcement Learning(강화 학습)에서 최적의 정책을 찾는 데 사용되는 개념 중 하나로, 함수는 다음과 같이 정의된다. $V^*(s) = \max_a \sum_{s', r} (P(s' | s, a) [r + \gamma V^*(s')])$, s 는 현재 상태, a 는 현재 상태에서 취할 수 있는 action, s' 는 다음 상태, r 은 보상, $P(s' | s, a)$ 은 주어진 상태 s 에서 행동 a 를 취했을 때 다음 상태가 s' 이고 보상이 r 일 확률, γ 은 discount factor 로 미래의 보상을 현재 가치로 얼마나 강조할지를 결정한다.

3. Result

```
[[-200  -1 -100  -1  -1  -1  -1]
 [  -1  -1 -100  -1  -1  -1  -1]
 [  -1  -1  -1  -1  -1  -1  -1]
 [  -1  -1  -1  -1 -100 -100  -1]
 [  -1  -1  -1  -1  -1  -1  -1]
 [  -1  -1  -1  -1  -1  -1  -1]
 [  -1  -1 -100 -100  -1  -1   0]]
```

7x7 크기의 시작점과 끝점을 가진 grid-world를 구현한 후 출력하였다. 우선 7x7 크기의

grid를 생성한 후 모든 값을 -1로 초기화 하였다. 그 후 시작점은 -100보다 작은 값으로 주기 위해 -200을 주었고(시작점으로 가지 않게 하기 위해) 끝점은 0으로 표기하였으며, 중간중간 -100은 함정에 해당한다.

7x7 그리드의 각 위치에서 최적 행동을 계산하는 함수를 따로 구현하였다. 각 위치에서 가능한 움직임 중 현재 가치가 가장 높은 방향을 찾아 최적 action으로 결정하고, 특정 위치에서 움직일 수 없는 경우 해당 방향은 제외하였고 결과를 2D 배열에 저장하도록 구현하였다.

- Policy evaluation

Grid world 에 대한 random policy evaluation 을 수행하는 함수를 코드로 구현하였다. 주변 값의 평균을 계산하여 현재 위치의 value 를 업데이트 하고, 주어진 반복 횟수 동안 위의 작업을 반복하였다. 실행 결과는 아래와 같다.

```
Iteration 0:
[[0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]]

Iteration 1:
[[-200. -1. -100. -1. -1. -1. -1.]
 [ -1. -1. -100. -1. -1. -1. -1.]
 [ -1. -1. -1. -1. -1. -1. -1.]
 [ -1. -1. -1. -1. -100. -100. -1.]
 [ -1. -1. -1. -1. -1. -1. -1.]
 [ -1. -1. -1. -1. -1. -1. -1.]
 [ -1. -1. -100. -100. -1. -1. 0.]]

Iteration 2:
[[-300.5 -76.5 -150.5 -26.8 -2. -2. -2.]
 [-51.8 -26.8 -125.8 -26.8 -2. -2. -2.]
 [ -2. -2. -26.8 -2. -26.8 -26.8 -2.]
 [ -2. -2. -2. -26.8 -125.8 -125.8 -26.8]
 [ -2. -2. -2. -2. -26.8 -26.8 -2.]
 [ -2. -2. -26.8 -26.8 -2. -2. -1.8]
 [ -2. -26.8 -150.5 -150.5 -26.8 -1.8 0.]]

Iteration 3:
[[-382.3 -139.6 -194.9 -52.5 -9.2 -3. -3.]
 [-96.3 -65. -157.7 -40.2 -15.4 -9.2 -3.]
 [-15.4 -15.4 -34. -27.8 -40.2 -40.2 -15.4]
 [ -3. -3. -15.4 -34. -151.6 -151.5 -40.2]
 [ -3. -3. -9.2 -21.6 -40.2 -40.2 -15.4]
 [ -3. -15.4 -46.3 -46.3 -21.6 -9.1 -2.4]
 [-9.2 -46.3 -188.6 -188.6 -46.3 -8.6 0.]]

Iteration 4:
[[-450.1 -196.4 -236.2 -75.2 -21. -7.1 -4.]
 [-140.8 -103.2 -183.5 -64.4 -25.7 -16.4 -8.6]
 [-33.5 -30.4 -55.1 -38.1 -59.8 -55.1 -25.7]
 [ -7.1 -10.2 -21. -55.1 -166.5 -168. -56.6]
 [ -4. -8.6 -22.6 -33.4 -59.8 -55. -25.6]
 [ -8.6 -25.6 -65.9 -70.5 -36.5 -19.2 -7.7]
 [-17.9 -65.9 -217.4 -217.4 -67.3 -17. 0.]]

Iteration 5:
[[-509.4 -247.5 -272.8 -100.2 -33.2 -13.1 -6.9]
 [-182.9 -138.8 -214.7 -81.6 -41.4 -25.1 -14.7]
 [-54. -51.5 -69.2 -59.6 -72.4 -68.5 -37.5]
 [-14.7 -17.8 -36.8 -65.8 -185.7 -183.3 -70.]
 [-8.1 -16.6 -33.2 -53. -73.8 -69.2 -37.2]
 [-15. -38.2 -85. -89.3 -55.2 -30. -14.1]
 [-28.6 -82.7 -241.6 -243.2 -85.6 -26.9 0.]]

Iteration 1994:
[[-5368.8 -5014.6 -4876.8 -4501.9 -4236.2 -4055.8 -3960.6]
 [-4923.2 -4794.2 -4713.8 -4388.8 -4147.1 -3966.8 -3861.5]
 [-4602.8 -4521.2 -4395.6 -4188.4 -3992.6 -3798.8 -3653.2]
 [-4360.2 -4288.3 -4155.2 -3972.8 -3832. -3578.7 -3295.5]
 [-4185.7 -4112.6 -3960.3 -3711.7 -3383.7 -2988.6 -2650.7]
 [-4080.4 -4012.3 -3857.7 -3526.1 -2998.6 -2337.5 -1664. ]
 [-4039.4 -3994.6 -3928.2 -3532.5 -2743.3 -1694.9 0.]]

Iteration 1995:
[[-5368.8 -5014.6 -4876.8 -4501.9 -4236.2 -4055.8 -3960.6]
 [-4923.2 -4794.2 -4713.8 -4388.8 -4147.1 -3966.8 -3861.5]
 [-4602.8 -4521.2 -4395.6 -4188.4 -3992.6 -3798.8 -3653.2]
 [-4360.2 -4288.3 -4155.2 -3972.8 -3832. -3578.7 -3295.5]
 [-4185.7 -4112.6 -3960.3 -3711.7 -3383.7 -2988.6 -2650.7]
 [-4080.4 -4012.3 -3857.7 -3526.1 -2998.6 -2337.5 -1664. ]
 [-4039.4 -3994.6 -3928.2 -3532.5 -2743.3 -1694.9 0.]]

Iteration 1996:
[[-5368.8 -5014.6 -4876.8 -4501.9 -4236.2 -4055.8 -3960.6]
 [-4923.2 -4794.2 -4713.8 -4388.8 -4147.1 -3966.8 -3861.5]
 [-4602.8 -4521.2 -4395.6 -4188.4 -3992.6 -3798.8 -3653.2]
 [-4360.2 -4288.3 -4155.2 -3972.8 -3832. -3578.7 -3295.5]
 [-4185.7 -4112.6 -3960.3 -3711.7 -3383.7 -2988.6 -2650.7]
 [-4080.4 -4012.3 -3857.7 -3526.1 -2998.6 -2337.5 -1664. ]
 [-4039.4 -3994.6 -3928.2 -3532.5 -2743.3 -1694.9 0.]]

Iteration 1997:
[[-5368.8 -5014.6 -4876.8 -4501.9 -4236.2 -4055.8 -3960.6]
 [-4923.2 -4794.2 -4713.8 -4388.8 -4147.1 -3966.8 -3861.5]
 [-4602.8 -4521.2 -4395.6 -4188.4 -3992.6 -3798.8 -3653.2]
 [-4360.2 -4288.3 -4155.2 -3972.8 -3832. -3578.7 -3295.5]
 [-4185.7 -4112.6 -3960.3 -3711.7 -3383.7 -2988.6 -2650.7]
 [-4080.4 -4012.3 -3857.7 -3526.1 -2998.6 -2337.5 -1664. ]
 [-4039.4 -3994.6 -3928.2 -3532.5 -2743.3 -1694.9 0.]]

Iteration 1998:
[[-5368.8 -5014.6 -4876.8 -4501.9 -4236.2 -4055.8 -3960.6]
 [-4923.2 -4794.2 -4713.8 -4388.8 -4147.1 -3966.8 -3861.5]
 [-4602.8 -4521.2 -4395.6 -4188.4 -3992.6 -3798.8 -3653.2]
 [-4360.2 -4288.3 -4155.2 -3972.8 -3832. -3578.7 -3295.5]
 [-4185.7 -4112.6 -3960.3 -3711.7 -3383.7 -2988.6 -2650.7]
 [-4080.4 -4012.3 -3857.7 -3526.1 -2998.6 -2337.5 -1664. ]
 [-4039.4 -3994.6 -3928.2 -3532.5 -2743.3 -1694.9 0.]]

Iteration 1999:
[[-5368.8 -5014.6 -4876.8 -4501.9 -4236.2 -4055.8 -3960.6]
 [-4923.2 -4794.2 -4713.8 -4388.8 -4147.1 -3966.8 -3861.5]
 [-4602.8 -4521.2 -4395.6 -4188.4 -3992.6 -3798.8 -3653.2]
 [-4360.2 -4288.3 -4155.2 -3972.8 -3832. -3578.7 -3295.5]
 [-4185.7 -4112.6 -3960.3 -3711.7 -3383.7 -2988.6 -2650.7]
 [-4080.4 -4012.3 -3857.7 -3526.1 -2998.6 -2337.5 -1664. ]
 [-4039.4 -3994.6 -3928.2 -3532.5 -2743.3 -1694.9 0.]]

Final Result (Iteration 2000):
[[-5368.8 -5014.6 -4876.8 -4501.9 -4236.2 -4055.8 -3960.6]
 [-4923.2 -4794.2 -4713.8 -4388.8 -4147.1 -3966.8 -3861.5]
 [-4602.8 -4521.2 -4395.6 -4188.4 -3992.6 -3798.8 -3653.2]
 [-4360.2 -4288.3 -4155.2 -3972.8 -3832. -3578.7 -3295.5]
 [-4185.7 -4112.6 -3960.3 -3711.7 -3383.7 -2988.6 -2650.7]
 [-4080.4 -4012.3 -3857.7 -3526.1 -2998.6 -2337.5 -1664. ]
 [-4039.4 -3994.6 -3928.2 -3532.5 -2743.3 -1694.9 0.]]
```

Iteration 이 k 와 같으며, 왼쪽 사진이 초반에 해당한다. K 가 0 일 때 초기화한 0 으로 되어 있는 것을 볼 수 있으며, k 가 1 일 때 grid world 를 기반으로 반복하며 업데이트 하는 것을 확인할 수 있다. 오른쪽 사진은 k 의 값이 수렴하게 되는 것을 나타낸 사진이다. 처음에는 큰 숫자를 넣어 반복해보았는데, 2000 번 반복해본 결과 값이 변하지 않고 거의 일치하며 수렴하는 것을 확인할 수 있었다. 약 1000 번 반복부터 소수점 단위로 변하는 부분이 존재하며 변하는 폭이 매우 작았다. 하지만 완벽하게 값이 수렴하는 반복은 약 2000 번 실행했을 때부터 인 것을 확인하였다.

















































```
[['list(['Down']) list(['Down']) list(['Right']) list(['Right'])
list(['Right']) list(['Right']) list(['Down'])]
list(['Down']) list(['Down']) list(['Right']) list(['Right'])
list(['Right']) list(['Down']) list(['Down'])]
list(['Down']) list(['Down']) list(['Down']) list(['Down'])
list(['Right']) list(['Down']) list(['Down'])]
list(['Down']) list(['Down']) list(['Down']) list(['Down'])
list(['Down']) list(['Down']) list(['Down'])]
list(['Down']) list(['Right']) list(['Right']) list(['Right'])
list(['Right']) list(['Down']) list(['Down'])]
list(['Right']) list(['Right']) list(['Right']) list(['Right'])
list(['Right']) list(['Right']) list(['Down'])]
list(['Right']) list(['Right']) list(['Right']) list(['Right'])
list(['Right']) list(['Right']) list(['0'])]
```

















































↓	↓	→	→	→	→	↓
↓	↓	→	→	→	↓	↓
↓	↓	↓	↓	→	↓	↓
↓	↓	↓	↓	↓	↓	↓
↓	→	→	→	→	↓	↓
→	→	→	→	→	→	↓
→	→	→	→	→	→	0

Random policy 를 최적화 한 결과를 출력해보았다.

























































- Policy Improvement

Iteration 을 반복하며 true value function 을 (1)에서 찾았으므로 해당 policy 를 따르는 것이 좋을지 안 좋을지 판단을 하는 부분이며, Greedy policy improvement 로 구현하였다. 각 상태에서 현재 policy 에 따라 최적의 action 을 계산하고 해당 action 들을 기반으로 policy 를 업데이트 하였다. 아래는 policy 가 업데이트 되는 과정과 각 state 에서의 action 을 나타낸 결과화면이다.
































































						
						
						
						
						
						
						

K 가 1 일 때의 policy 가 update 된 결과를 볼 수 있다.

```
Iteration 3:
[[-202, -3, -102, -3, -3, -3, -3]]
[[-3, -3, -102, -3, -3, -3, -3]]
[[-3, -3, -3, -3, -3, -3, -3]]
[[-3, -3, -3, -3, -102, -102, -3]]
[[-3, -3, -3, -3, -3, -3, -2]]
[[-3, -3, -3, -3, -3, -2, -1]]
[[-3, -3, -102, -102, -2, -1, 0]]

[[list(['Down', 'Right']) list(['Down']) list(['Left', 'Right'])
list(['Down', 'Right']) list(['Down', 'Left', 'Right'])
list(['Down', 'Left', 'Right']) list(['Down', 'Left', 'Right'])]]
[[list(['Down', 'Right']) list(['Up', 'Down', 'Left'])
list(['Down', 'Left', 'Right']) list(['Up', 'Down', 'Right'])
list(['Up', 'Down', 'Left', 'Right'])
list(['Up', 'Down', 'Left'])]]
[[list(['Up', 'Down', 'Right']) list(['Up', 'Down', 'Left', 'Right'])
list(['Up', 'Down', 'Right']) list(['Up', 'Down', 'Left', 'Right'])
list(['Up', 'Left', 'Right']) list(['Up', 'Left', 'Right'])
list(['Up', 'Down', 'Left'])]]
[[list(['Up', 'Down', 'Right']) list(['Up', 'Down', 'Left', 'Right'])
list(['Up', 'Down', 'Left', 'Right']) list(['Up', 'Down', 'Left'])
list(['Up', 'Down', 'Left']) list(['Up', 'Down', 'Right'])
list(['Down'])]]
[[list(['Up', 'Down', 'Right']) list(['Up', 'Down', 'Left', 'Right'])
list(['Up', 'Down', 'Left', 'Right'])
list(['Up', 'Down', 'Left', 'Right']) list(['Down', 'Left', 'Right'])
list(['Down', 'Right']) list(['Down'])]]
[[list(['Up', 'Down', 'Right') list(['Up', 'Down', 'Left', 'Right'])
list(['Up', 'Left', 'Right') list(['Up', 'Left', 'Right'])
list(['Down', 'Right') list(['Down', 'Right']) list(['Down'])]]
[[list(['Up', 'Left', 'Right') list(['Up', 'Left') list(['Up', 'Left'])
list(['Right') list(['Right') list(['Right') list(['Right')]]]
```

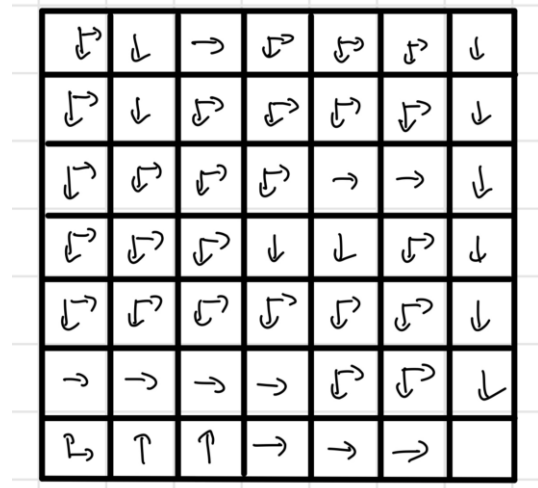
						
						
						
						
						
						
						
						
						

K 가 3 일 때의 결과 역시 종료점에 최단거리로 가는 경로가 적용되는 것을 확인할 수 있다.

```

Iteration 11:
[[-210, -11, -109, -9, -8, -7, -6.]
 [-11, -10, -108, -8, -7, -6, -5.]
 [-10, -9, -8, -7, -6, -5, -4.]
 [-9, -8, -7, -6, -104, -103, -3.]
 [-8, -7, -6, -5, -4, -3, -2.]
 [-7, -6, -5, -4, -3, -2, -1.]
 [-8, -7, -105, -102, -2, -1, 0.]]
[[list(['Down', 'Right']) list(['Down']) list(['Right'])]
 list(['Down', 'Right']) list(['Down', 'Right']) list(['Down', 'Right'])
 list(['Down'])]]
[[list(['Down', 'Right']) list(['Down']) list(['Down', 'Right'])]
 list(['Down', 'Right']) list(['Down', 'Right']) list(['Down', 'Right'])
 list(['Down'])]]
[[list(['Down', 'Right']) list(['Down', 'Right']) list(['Down', 'Right'])]
 list(['Down', 'Right']) list(['Right']) list(['Right']) list(['Down'])]]
[[list(['Down', 'Right']) list(['Down', 'Right']) list(['Down', 'Right'])]
 list(['Down']) list(['Down']) list(['Down', 'Right']) list(['Down'])]]
[[list(['Down', 'Right']) list(['Down', 'Right']) list(['Down', 'Right'])]
 list(['Down', 'Right']) list(['Down', 'Right']) list(['Down', 'Right'])
 list(['Down'])]]
[[list(['Right']) list(['Right']) list(['Right']) list(['Right'])]
 list(['Down', 'Right']) list(['Down', 'Right']) list(['Down'])]]
[[list(['Up', 'Right']) list(['Up']) list(['Up']) list(['Up'])]
 list(['Right']) list(['Right']) list(['Right']) list(['Right'])]]

```

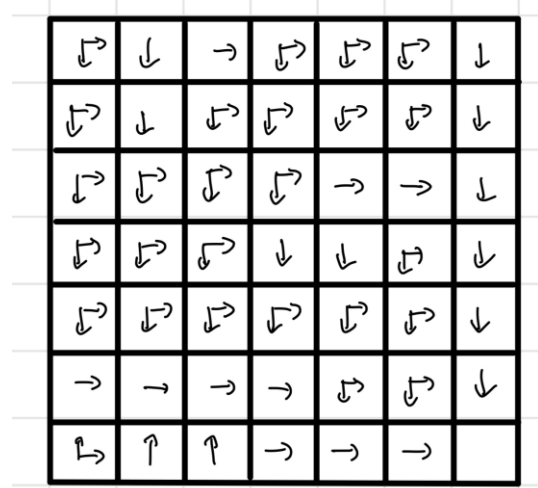


K 가 11 일 때의 결과이다.

```

Iteration 12:
[[-211, -11, -109, -9, -8, -7, -6.]
 [-11, -10, -108, -8, -7, -6, -5.]
 [-10, -9, -8, -7, -6, -5, -4.]
 [-9, -8, -7, -6, -104, -103, -3.]
 [-8, -7, -6, -5, -4, -3, -2.]
 [-7, -6, -5, -4, -3, -2, -1.]
 [-8, -7, -105, -102, -2, -1, 0.]]
[[list(['Down', 'Right']) list(['Down']) list(['Right'])]
 list(['Down', 'Right']) list(['Down', 'Right']) list(['Down', 'Right'])
 list(['Down'])]]
[[list(['Down', 'Right']) list(['Down']) list(['Down', 'Right'])]
 list(['Down', 'Right']) list(['Down', 'Right']) list(['Down', 'Right'])
 list(['Down'])]]
[[list(['Down', 'Right']) list(['Down', 'Right']) list(['Down', 'Right'])]
 list(['Down', 'Right']) list(['Right']) list(['Right']) list(['Down'])]]
[[list(['Down', 'Right']) list(['Down', 'Right']) list(['Down', 'Right'])]
 list(['Down']) list(['Down']) list(['Down', 'Right']) list(['Down'])]]
[[list(['Down', 'Right']) list(['Down', 'Right']) list(['Down', 'Right'])]
 list(['Down', 'Right']) list(['Down', 'Right']) list(['Down', 'Right'])
 list(['Down'])]]
[[list(['Right']) list(['Right']) list(['Right']) list(['Right'])]
 list(['Down', 'Right']) list(['Down', 'Right']) list(['Down'])]]
[[list(['Up', 'Right']) list(['Up']) list(['Up']) list(['Up'])]
 list(['Right']) list(['Right']) list(['Right']) list(['Right'])]]

```



마지막으로 k가 12일 때의 결과로, 값이 수렴한 것을 확인할 수 있으며 optimal policy에 해당하는 것을 알 수 있다.

20 번 정도 iteration 해봤는데 12 번째 반복부터 값이 동일한 것을 확인할 수 있었고 이로써 optimal policy 를 구할 수 있었다.

그럼 policy Improvement 를 구현해봤는데, random policy 와 greedy policy 의 차이를 비교해보아 otimal policy 를 찾아보겠다. 우선 random policy 는 2000 번의 반복 후에 optimal policy 를 찾을 수 있었으며 greedy policy 의 경우 12 번의 반복 후 찾을 수 있었다. 두 방법 모두 종료점까지의 최단거리를 찾을 수 있었지만 실행횟수에서 큰 차이를 보이는 것을 확인할 수 있다. Random policy 의 경우 매 반복마다 update 를 하기 때문에 함정으로 가는 경우가 고려되지 않는다. 이로 인해 함정의 reward 값이 낮아질때까지 반복이 필요하기 때문에 반복횟수가 크게 증가하게 된다. greedy policy 의

경우 함정에 가지 않고 update 를 하기 때문에 random policy 보다 실행횟수가 적게 나오는 것은 당연한 결과를 볼 수 있었다. 따라서 greedy policy 를 이용하여 구한 policy 가 optimal policy 라고 할 수 있다.

- Value Iteration

마지막으로 Value Iteration 부분이다. 주어진 grid world 의 각 state 에 대한 최적 가치를 계산하는 함수를 구현했으며, 각 state 에서 주변 상태의 value 를 고려해 현재 state 의 value 를 업데이트 하도록 구현하였다.

```

Iteration 0:
[[0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]]

Iteration 1:
[[-200. -1. -100. -1. -1. -1. -1.]
 [-1. -1. -100. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -100. -100. -1.]
 [-1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -100. -100. -1. -1. 0.]]

Iteration 2:
[[-201. -2. -101. -2. -2. -2. -2.]
 [-2. -2. -101. -2. -2. -2. -2.]
 [-2. -2. -2. -2. -2. -2. -2.]
 [-2. -2. -2. -2. -101. -101. -2.]
 [-2. -2. -2. -2. -2. -2. -2.]
 [-2. -2. -2. -2. -2. -2. -1.]
 [-2. -2. -101. -101. -2. -1. 0.]]

Iteration 3:
[[-202. -3. -102. -3. -3. -3. -3.]
 [-3. -3. -102. -3. -3. -3. -3.]
 [-3. -3. -3. -3. -3. -3. -3.]
 [-3. -3. -3. -3. -102. -102. -3.]
 [-3. -3. -3. -3. -3. -3. -2.]
 [-3. -3. -3. -3. -3. -2. -1.]
 [-3. -3. -102. -102. -2. -1. 0.]]

Iteration 4:
[[-203. -4. -103. -4. -4. -4. -4.]
 [-4. -4. -103. -4. -4. -4. -4.]
 [-4. -4. -4. -4. -4. -4. -4.]
 [-4. -4. -4. -4. -103. -103. -3.]
 [-4. -4. -4. -4. -4. -3. -2.]
 [-4. -4. -4. -4. -3. -2. -1.]
 [-4. -4. -103. -102. -2. -1. 0.]]

Iteration 5:
[[-204. -5. -104. -5. -5. -5. -5.]
 [-5. -5. -104. -5. -5. -5. -5.]
 [-5. -5. -5. -5. -5. -5. -4.]
 [-5. -5. -5. -5. -104. -103. -3.]
 [-5. -5. -5. -5. -4. -3. -2.]
 [-5. -5. -5. -4. -3. -2. -1.]
 [-5. -5. -104. -102. -2. -1. 0.]]

Iteration 10:
[[-209. -10. -109. -9. -8. -7. -6.]
 [-10. -10. -108. -8. -7. -6. -5.]
 [-10. -9. -8. -7. -6. -5. -4.]
 [-9. -8. -7. -6. -104. -103. -3.]
 [-8. -7. -6. -5. -4. -3. -2.]
 [-7. -6. -5. -4. -3. -2. -1.]
 [-8. -7. -105. -102. -2. -1. 0.]]

Iteration 11:
[[-210. -11. -109. -9. -8. -7. -6.]
 [-11. -10. -108. -8. -7. -6. -5.]
 [-10. -9. -8. -7. -6. -5. -4.]
 [-9. -8. -7. -6. -104. -103. -3.]
 [-8. -7. -6. -5. -4. -3. -2.]
 [-7. -6. -5. -4. -3. -2. -1.]
 [-8. -7. -105. -102. -2. -1. 0.]]

Iteration 12:
[[-211. -11. -109. -9. -8. -7. -6.]
 [-11. -10. -108. -8. -7. -6. -5.]
 [-10. -9. -8. -7. -6. -5. -4.]
 [-9. -8. -7. -6. -104. -103. -3.]
 [-8. -7. -6. -5. -4. -3. -2.]
 [-7. -6. -5. -4. -3. -2. -1.]
 [-8. -7. -105. -102. -2. -1. 0.]]

Iteration 13:
[[-211. -11. -109. -9. -8. -7. -6.]
 [-11. -10. -108. -8. -7. -6. -5.]
 [-10. -9. -8. -7. -6. -5. -4.]
 [-9. -8. -7. -6. -104. -103. -3.]
 [-8. -7. -6. -5. -4. -3. -2.]
 [-7. -6. -5. -4. -3. -2. -1.]
 [-8. -7. -105. -102. -2. -1. 0.]]

Iteration 14:
[[-211. -11. -109. -9. -8. -7. -6.]
 [-11. -10. -108. -8. -7. -6. -5.]
 [-10. -9. -8. -7. -6. -5. -4.]
 [-9. -8. -7. -6. -104. -103. -3.]
 [-8. -7. -6. -5. -4. -3. -2.]
 [-7. -6. -5. -4. -3. -2. -1.]
 [-8. -7. -105. -102. -2. -1. 0.]]

Iteration 15:
[[-211. -11. -109. -9. -8. -7. -6.]
 [-11. -10. -108. -8. -7. -6. -5.]
 [-10. -9. -8. -7. -6. -5. -4.]
 [-9. -8. -7. -6. -104. -103. -3.]
 [-8. -7. -6. -5. -4. -3. -2.]
 [-7. -6. -5. -4. -3. -2. -1.]
 [-8. -7. -105. -102. -2. -1. 0.]]

Iteration 16:
[[-211. -11. -109. -9. -8. -7. -6.]
 [-11. -10. -108. -8. -7. -6. -5.]
 [-10. -9. -8. -7. -6. -5. -4.]
 [-9. -8. -7. -6. -104. -103. -3.]
 [-8. -7. -6. -5. -4. -3. -2.]
 [-7. -6. -5. -4. -3. -2. -1.]
 [-8. -7. -105. -102. -2. -1. 0.]]

```

약 20 번 반복하였는데, 12 번 반복부터 결과가 수렴하는 것을 볼 수 있으며, (2)에서 확인하였던 것 처럼 Greedy policy 와 동일한 결과를 보이는 것을 확인할 수 있다.

4. Consideration

강의와 강의자료를 바탕으로 진행되었던 과제이기 때문에 개념을 이해하는데 있어서는 큰 어려움이 없었다. 하지만 이론상 이해한 점을 코드로 구현하는 과정에서 좀 힘들었던 것 같다. 우선 파이썬을 많이 사용해보지 않아 파이썬으로 구현하는 점이 조금 까다로웠다. 내가 원하는 대로 흘러가지 않아 구글링을 많이 해보며 여러 함수들을 사용하여 해결하였다. 또한 알고리즘 시간에 배웠던 dynamic programming 을 인공지능 시간에도 활용하여 과제를 해결하니 신기했고 과제를 해결해 나가는 과정이 subproblem 을 풀어나가는 과정인 것을 깨닫고 아 이렇게 응용될 수 있구나 하는 생각도 들었다. 과제를 구현하면서 어려웠던 부분은 아무래도 두번째 부분인 것 같다. 첫 번째와 세 번째 과제는 평균을 내거나 max 값을 뽑아 내는 것이 다였는데, greedy policy 의 경우 방향성도 고려하여 구현해야 하다 보니 결과도 잘 안 나왔고 다소 시간이 오래 걸렸다. 하지만 결과에 정답은 없다고 생각했고 주변 친구들과 결과는 조금 달랐지만 optimal policy 를 찾는 iteration 의 횟수는 동일했고 무엇보다 목적지에 도달할 수 있다는 점에서 과제를 성공적으로 해냈다고 생각한다. 과제를 1 차적으로 제출한 후 끝점으로만 가게 구현하지 않아 다시 수정하였다. 처음에는 시작점으로도 갈 수 있게 구현했기 때문에 시작점으로 가지 않도록 하기 위해 시작점에 -200 을 넣어주었으며 끝점만 0 으로 설정하여 시작점도 계산에 포함되도록 코드를 바꿔주었다. 또한 제일 바깥쪽 state 에서 grid-world 밖으로 나가는 action 을 취할 경우 제자리로 돌아오기 때문에 (2)의 과정에서 matrix를 그릴 때 해당 경우를 생략하여 표현하였다. Ex) [1, 0]의 action 이 down, left, right 일 경우 down, right 만 표시함