인공지능

Project 3

학　　과: 컴퓨터정보공학부

담당교수: 박철수 교수님

학　　번: 2019202050

성　　명: 이강현

1. Introduction

이번 프로젝트에서는 Dynamic Programming 을 이용하여 제공된 grid world 를 이용하여 policy evaluation 을 진행하여 policy 를 prediction 해보고 그를 토대로 policy iteration 과 value iteration 을 진행하여 optimal policy 를 비교해본다.

2. Algorithm

Dynamic Programming: 크고 복잡한 문제를 여러 부분 문제로 나누어 해를 구하고 이러한 해를 이용하여 original 한 문제를 해결하는 방법 optimal substructure 를 가지고 부분문제들을 구하기 위한 재귀적인 실행 중에 부분문제들의 해가 서로 겹치는 경우 사용하면 유용한 방법이다.

Policy evaluation: random policy 로 policy 를 고정하고 매 step 마다 state 의 value function 을 update 한다. 이를 토대로 random policy 의 true value function 을 구할 수 있다. update 는 Bellman Expectation Equation 를 이용하여 계산한다.

policy improvement: update 한 value function 을 기반으로 현 시점에서의 더 나은 policy 를 찾아내는 것을 말한다.

policy iteration: policy evaluation 과 policy improvement 를 반복하여 optimal policy 를 얻어내는 과정을 말한다.

3. Result

먼저 코드를 구현하기 전에 value update 의 기준이 되는 grid world 를 먼저 7x7 배열로 만들어 두어 사용하였다.

```
[[-101   -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]]
```

이 grid world 는 해당 state 에서 agent 가 어떤 액션을 취했을 때 받게 되는 reward 를 가지고 있다. 따라서 대부분에 경우에는 -1 의 reward 를 움직일 때마다 받게 되지만 함정에 다다르게 되면 -100 의 reward 를 받게 된다. 또한 출발점으로부터 도착지까지의 최단거리를 구하기 위해서 출발점의 reward 는 함정을 만나서 다시 돌아오는 경우가 생기지 않게 함정의 reward 보다 더 낮은 값을 주었고 도착지점은 가장 높은 reward 를 받게끔 구성하였다.

<policy evaluation>

먼저 random policy 로 고정하고 policy evaluation 을 진행하였다. 많은 반복을 통해서 1000 회 정도 반복한 결과 값들이 수렴하여 변하는 폭이 매우 적음을 알게 되었다. 따라서 1000 회 반복으로 마무리하였다.

```
k = 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.]]
k = 1
[[-101.   -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.]]
k = 2
[[-152.   -51.8 -150.5  -26.8   -2.    -2.    -2. ]
 [ -27.   -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. ]]
k = 3
[[-196.7  -96.3 -188.7  -52.5   -9.2   -3.    -3. ]
 [ -53.   -52.6 -157.7  -40.2  -15.4   -9.2   -3. ]
 [  -9.2  -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. ]]
```

먼저 k=0,1,2,3 일 때 반복 초반에 변하는 부분이다. 처음에는 모두 0 으로 초기화되어 있었고 그 이후로는 grid world 를 이용하여 value 를 update 하는 것을 확인할 수 있다. k=1->2 과정에서 행: 1, 열: 3 에 해당하는 부분을 예시로 설명한다. k=2 일 때의 해당 state 를 update 하는

것이기 때문에 k=1 일때의 상하좌우 value 를 이용할 것이다. 이동할 때마다 액션을 취했을 때 받게 될 reward 는 grid world 에서의 현재 state 를 더해주었다. 따라서 왼쪽으로 갈 때 (-100 - 1), 오른쪽을 갈 때 (-100 - 1), 아래로 갈 때 (-100 – 100), 위로 갈 때 위의 공간은 없기 때문에 자기 자신의 값을 참조하여 (-100 – 100)이 된다. 이를 모두 더하고 random policy 에 따라 0.25 를 곱해주게 되면 -150.5 가 나오게 된다.

```
k = 998
[[-4319.  -4163.1 -4145.  -3851.  -3641.8 -3498.4 -3421.8]
 [-4071.9 -4022.2 -4021.6 -3763.  -3572.8 -3428.4 -3341.7]
 [-3871.2 -3829.  -3756.8 -3603.2 -3454.7 -3297.1 -3171.7]
 [-3709.6 -3662.6 -3570.1 -3435.2 -3342.2 -3130.3 -2872.8]
 [-3591.6 -3538.6 -3422.6 -3222.1 -2949.3 -2609.7 -2313. ]
 [-3523.4 -3474.2 -3356.2 -3077.8 -2619.7 -2042.5 -1453.1]
 [-3501.  -3475.1 -3446.9 -3110.  -2405.7 -1484.      0. ]]
k = 999
[[-4319.2 -4163.3 -4145.2 -3851.2 -3642.  -3498.6 -3421.9]
 [-4072.1 -4022.4 -4021.8 -3763.2 -3573.  -3428.5 -3341.9]
 [-3871.4 -3829.2 -3757.  -3603.4 -3454.8 -3297.3 -3171.8]
 [-3709.8 -3662.8 -3570.3 -3435.4 -3342.4 -3130.4 -2873. ]
 [-3591.8 -3538.8 -3422.8 -3222.2 -2949.4 -2609.8 -2313.1]
 [-3523.6 -3474.3 -3356.4 -3078.  -2619.8 -2042.6 -1453.2]
 [-3501.1 -3475.3 -3447.  -3110.1 -2405.8 -1484.      0. ]]
k = 1000
[[-4319.4 -4163.5 -4145.4 -3851.4 -3642.2 -3498.8 -3422.1]
 [-4072.3 -4022.6 -4022.  -3763.4 -3573.1 -3428.7 -3342. ]
 [-3871.6 -3829.4 -3757.2 -3603.6 -3455.  -3297.4 -3172. ]
 [-3710.  -3663.  -3570.5 -3435.6 -3342.5 -3130.6 -2873.1]
 [-3592.  -3538.9 -3422.9 -3222.4 -2949.6 -2609.9 -2313.3]
 [-3523.7 -3474.5 -3356.5 -3078.1 -2620.  -2042.7 -1453.2]
 [-3501.3 -3475.4 -3447.2 -3110.2 -2405.9 -1484.1     0. ]]
```

반복을 진행하다 보면 위와 같이 값이 수렴하여 거의 변하지 않는 것을 확인할 수 있다. 소수점 단위로 증가하는 폭이 적기 때문에 1000 번까지만 반복을 진행하고 이를 수렴된 값이라고 가정하였다.

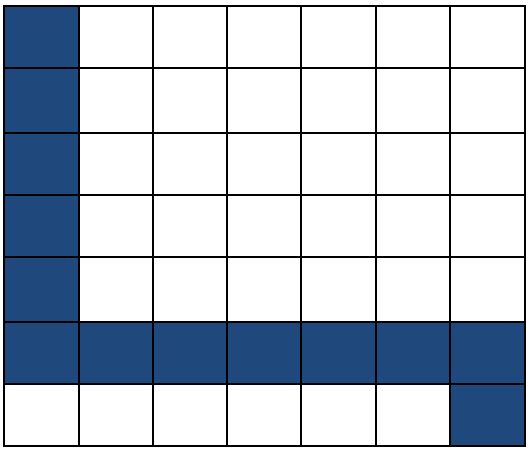
<policy improvement – random policy>

그럼 이제

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

수렴된 value 들을 이용하여 optimal policy 를 구해본다.

↓	↓	→	→	→	→	↓
↓	↓	↓	→	→	↓	↓
↓	↓	↓	↓	→	↓	↓
↓	↓	↓	↓	↓	↓	↓
↓	→	→	→	→	↓	↓
→	→	→	→	→	→	↓
→	→	→	→	→	→	End



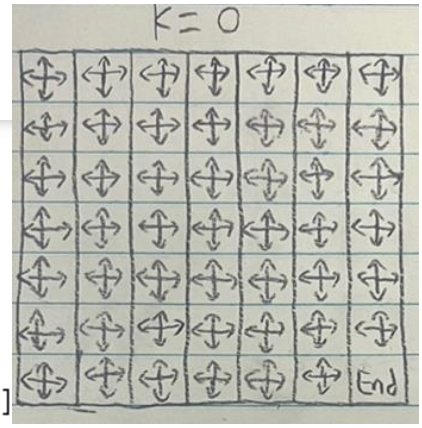
최종적으로 구해진 true value function 에 의해 policy 를 update 한 결과 위와 같이 구해진다.

출발지부터 policy 를 기반으로 이동해보면 최단거리로 함정을 잘 피하여 목적지로 이동하였다.

<policy improvement – greedy policy>

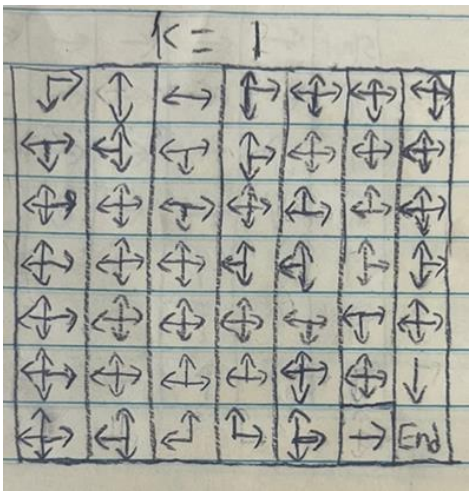
이번에는 매 state 마다 update 된 value 에 따라 policy 를 greedy 하게 update 하는 방식으로 policy 를 improvement 해본다.

```
k = 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.]]
```



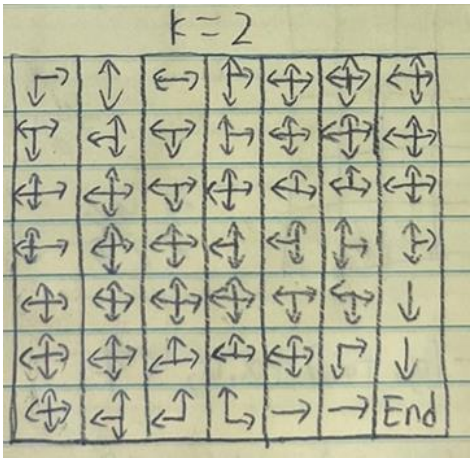
처음에는 모두 0 으로 초기화되어 있기 때문에 모든 방향으로의 policy 를 따르게 된다.

```
k = 1
[[-101.   -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.]]
```



한번 value 를 update 한 후에는 함정을 피하기 위해 혹은 목적지에 도달하기 위한 policy 가 value 값의 차이에 의해 적용되기 시작한다.

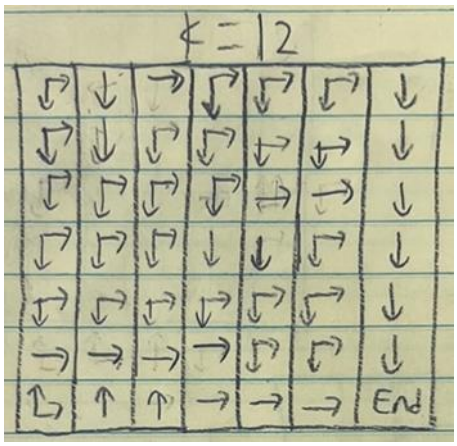
```
k = 2
[[-102.   -2. -101.   -2.   -2.   -2.   -2.]
 [  -2.   -2. -101.   -2.   -2.   -2.   -2.]
 [  -2.   -2.   -2.   -2.   -2.   -2.   -2.]
 [  -2.   -2.   -2.   -2. -101. -101.   -2.]
 [  -2.   -2.   -2.   -2.   -2.   -2.   -2.]
 [  -2.   -2.   -2.   -2.   -2.   -2.   -1.]
 [  -2.   -2. -101. -101.   -2.   -1.    0.]]
```



그 이후에는 목적지와 가까운 쪽 부분부터 더 높은 reward 를 갖게되면서 목적지에 최단거리로 가기 위한 방법이 policy 에 적용되기 시작한다.

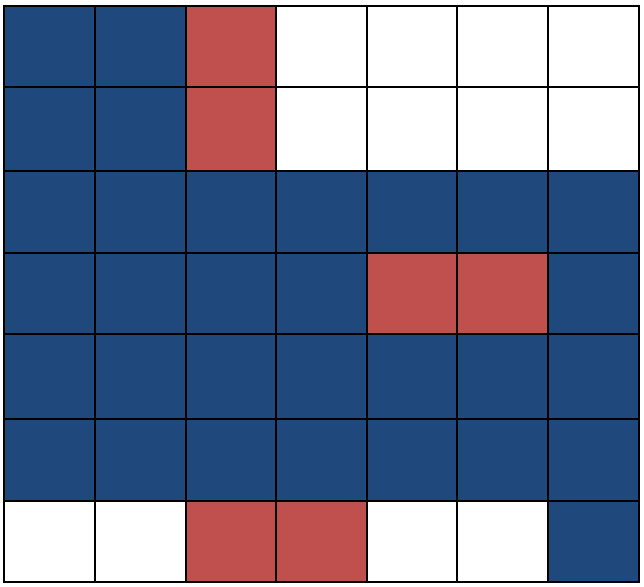
이를 반복하게 되면 k = 12 에 도달할 때 값이 거의 수렴하게 된다.

```
k = 12
[[-112.  -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.]]
```



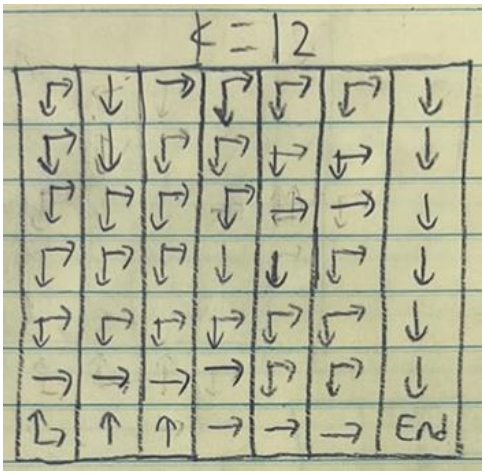
이로써 optimal policy 를 구하였다.

policy 를 이용한 최단거리는 다음과 같다.



파란색으로 칠해진 부분은 Policy 에 따라서 칠해진 공간만으로만 이동하게 되고 이 공간내에서 policy 를 따르면 최단거리로 목적지에 도달할 수 있게 된다. 빨간색으로 칠해진 부분은 함정이 있는 부분이고 함정에 도달하지 않게끔 이동 범위가 정해진 것을 확인할 수 있다.

그렇다면 policy 를 매 step 마다 update 한 greedy policy improvement 와 random policy 를 이용하여 최종적으로 구해진 value 를 이용하여 policy 를 update 한 경우의 차이는 어떨까?



<greedy policy improvement>

↓	↓	→	→	→	→	↓
↓	↓	↓	→	→	↓	↓
↓	↓	↓	↓	→	↓	↓
↓	↓	↓	↓	↓	↓	↓
↓	→	→	→	→	↓	↓
→	→	→	→	→	→	↓
→	→	→	→	→	→	End

<random policy improvement>

빨간색으로 칠한 부분은 함정을 나타내고 파란색으로 칠한 부분은 함정으로 가려는 경우이다. random policy improvement 방식은 policy 를 충실하게 따르면 파란색으로 칠해진 부분으로 가지 않긴 하지만 만약 가게 된다면 함정으로 가게끔 학습이 진행됐으나 greedy policy improvement 방식은 만약 파란색 부분으로 가게 되더라도 함정을 정상적으로 회피하게 된다. 두 방법 모두 목적지까지 최단거리로 가는 방법이 존재한다. 하지만 greedy policy improvement 방법과는 달리 random policy improvement 방법은 목적지로 도달하는 경로가 유일했다. 매 step 마다 value function 에는 함정이라는 것에 도달하였을 때 -100 이라는 작은 reward 를 받게끔 두 방법 모두 구현되었으나 policy 를 update 하여 사용한 greedy policy improvement 는 함정으로 이동하지 않는 반면 random policy improvement 는 함정으로 가는 경우도 고려하여 매 state 의 value function 을 update 하였기 때문이다. 결과적으로 이러한 방법은 value function 이 수렴하기까지의 실행 횟수에도 영향을 미쳤다. policy 를 매 step 마다 update 할 경우 value 가 작은 state 로 가는 방향은 고려되지 않기에 매 순간 최적의 방향만을 고려한다. 하지만 모든 방향을 고려하게 될 경우 오히려 반복 초기에는 함정이 고려되는 value function 을 가지지만 반복을 하면 할수록 함정의 reward 가 상하좌우의 state 를 update 하는데 영향을 미치기 때문에 value 를 update 하지 않는 목적지를 제외하고 전체적인 맵에 평준화되어버린다. 이러한 함정의 reward 값의 영향이 낮아질 때까지 반복을 진행해야 하기 때문에 수렴하기까지의 반복횟수가 많아질 수밖에 없다. 또한 함정의 reward 와 그렇지 않을 때의 reward 의 차이가 점점 감소하게 되면서 최단거리로 가는 경로의 수가 줄어든 것으로 예상한다. greedy 한 방법으로 구한 policy 가 grid world 를 고려했을 때 최단거리로 갈 수 있는 모든 경우의 수를 담고 있지는 않지만 더 많은 선택지를 제시했다는 점, 또 모든 state 에서 함정으로 가는 policy 가 하나도 없었다는 점을 고려하여 greedy policy improvement 를 optimal policy 라고 판단하였다.

<value iteration>

```
k = 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.]]
k = 1
[[-101.   -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.]]
k = 2
[[-102.   -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.]]
k = 3
[[-103.   -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.]]
```

....

```
k = 11
[[-111.  -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.]]
k = 12
[[-112.  -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.]]
k = 13
[[-112.  -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.]]
```

value iteration 의 value function update 과정은 greedy policy improvement 의 value function update 과정과 비슷하였다. greedy policy improvement 에서는 이전 state 에서 가장 높은 value 를 가진 state 쪽으로의 방향만 고려하여 현재 state 의 value 를 update 하였고 value iteration 에서는 모든 방향을 고려하지만 그중 최대 value function 만을 고려하니 결과적으로 value 가 update 되는 과정과 그 결과가 동일하게 나타났다.

4. Consideration

모델 free 한 상황에서 dynamic programming 을 이용하여 grid world 에서 policy 를 최적화시켜보았다. 7x7 이라는 environment 에서 최단거리를 구해내기 위해서는 여러 번 반복을 진행하여 어떤 것이 최단거리인지 모든 방향을 다 고려해야 알아낼 수 있다. 이는 복잡한 문제로 다가올 수 있는데 bellman equation 이 재귀적으로 구성되어 있다는 점에서 착안하여 여러 개의 subproblem 으로 나누고 각각에 대한 policy 를 최적화하게 된다면 하나의 state 에서는 단순히 현재 어느 방향으로 이동하는 게 좋다는 정보일 뿐이지만 이것이 최적이라면 전체적으로 보았을 때 최적이 되는 최적 구조를 만족하게 되어 최단거리를 구할 수 있게 되는 것이 흥미로웠다. 강화학습을 통해 reward 를 적절히 수정하면 내가 의도한 바에 최적으로 부합하는 어떠한 결과를 만들어낼 수 있다는 것을 배울 수 있었다.

5. Reference

인공지능 강의자료 참조