

4. 동적 계획법 (Dynamic Programming)

순천향대학교 컴퓨터공학과

이 상 정

4. 동적 계획법

학습 내용

- 동적 계획법 소개
- 정책 평가 (Policy Evaluation)
- 정책 개선 (Policy Iteration)
- 정책 반복 (Policy Iteration)
- 가치 반복 (Value Iteration)
- 그리드 월드 (Grid World) 예

동적 계획법 소개

4. 동적 계획법

학습과 계획

- 순차 의사 결정(sequential decision making)에는 **계획**과 **강화학습**과 두 가지 방식
- **계획 (Planning)**
 - **환경의 모델이 알려짐**, 모델 기반 에이전트
 - 에이전트 (상호작용 없이) **모델을 가지고 계산을 수행**
 - 에이전트는 정책을 개선
 - **동적 계획법 (dynamic programming)**이 이에 해당
- **강화학습 (Reinforcement Learning)**
 - 초기에 환경에 대해 알지 못함
 - 에이전트는 **환경과 상호작용**을 통해 환경을 파악
 - 에이전트는 정책을 생성하고 개선

- 동적 계획법 (Dynamic Programming)은 예측(prediction)과 제어(control) 두 단계로 구성
 - 강화학습은 동적 계획법을 기반으로 발전하였기 때문에 이의 이해가 중요
- 예측(prediction)은 주어진 정책을 사용하여 미래의 결과를 평가하고 행동하는 것
 - 현재의 최적화되지 않은 정책에 대해 가치함수를 계산
 - 정책 평가 (Policy Evaluation)
- 제어(control)는 가장 최적의 정책을 찾기 위해 최적화하는 것
 - 현재의 가치함수를 토대로 더 개선된 정책을 구하고, 이를 반복하여 최적의 정책을 도출
 - 정책 개선 (Policy Improvement)

정책 평가 (Policy Evaluation)

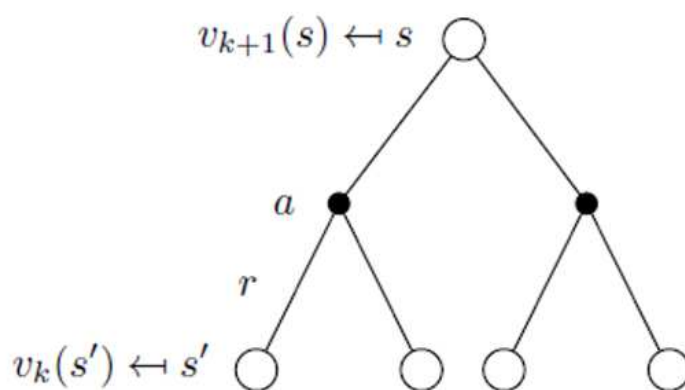
정책 평가 (Policy Evaluation) (1)

□ 벨만 기대 방정식 (Bellman Expectation Equation)을 사용하여 현재 주어진 정책에 대해 가치함수를 계산

- 한번에 모든 상태에 대해 계산하여 갱신
- 초기에는 어떤 정보도 없기 때문에 랜덤 정책(random policy)로 시작
- 무한히 반복하면 주어진 정책에 대한 **올바른 가치함수**를 계산
 - 이전에 계산된 예측값 $v_k(s')$ 로 부터 새로운 예측 값 $v_{k+1}(s)$ 를 계산
-> 이를 **부트스트랩(bootstrapping)** 기법이라 함

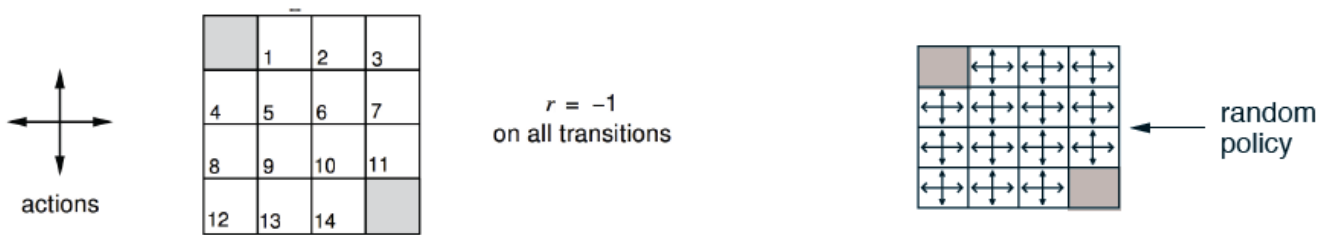
- At each iteration $k + 1$
- For all states $s \in \mathcal{S}$
- Update $v_{k+1}(s)$ from $v_k(s')$
- where s' is a successor state of s

정책 평가 (2)



$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) \left[r + \gamma v_k(s') \right]$$

정책 평가: 격자 예 (1)



- 14개의 비단말 상태 (nonterminal state)와 2개의 단말 상태 (terminal state)
- 상하 좌우 4개의 이동 가능한 행동
 - 에이전트가 특정 상태에서 특정 행동을 취할 때 전이될 상태의 확률은 1, 나머지 행동의 전이 확률은 0
- 모든 이동에 대해 할인율 1
- 그리드 밖으로 향한 행동 시 상태는 불변
- 비단말 상태 이동 시 보상은 -1
- 초기 정책은 균등한 랜덤 정책 (uniform random policy)

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) [r + \gamma v_k(s')]$$

$$\pi(n|\cdot) = \pi(e|\cdot) = \pi(s|\cdot) = \pi(w|\cdot) = 0.25$$

교 컴퓨터공학과

9

정책 평가: 격자 예 (2)

v_k for the Random Policy

$k = 0$	<table> <tr><td>0.0</td><td>0.0</td><td>0.0</td><td>0.0</td></tr> <tr><td>0.0</td><td>0.0</td><td>0.0</td><td>0.0</td></tr> <tr><td>0.0</td><td>0.0</td><td>0.0</td><td>0.0</td></tr> <tr><td>0.0</td><td>0.0</td><td>0.0</td><td>0.0</td></tr> </table>	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	$k = 3$	<table> <tr><td>0.0</td><td>-2.4</td><td>-2.9</td><td>-3.0</td></tr> <tr><td>-2.4</td><td>-2.9</td><td>-3.0</td><td>-2.9</td></tr> <tr><td>-2.9</td><td>-3.0</td><td>-2.9</td><td>-2.4</td></tr> <tr><td>-3.0</td><td>-2.9</td><td>-2.4</td><td>0.0</td></tr> </table>	0.0	-2.4	-2.9	-3.0	-2.4	-2.9	-3.0	-2.9	-2.9	-3.0	-2.9	-2.4	-3.0	-2.9	-2.4	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	-2.4	-2.9	-3.0																																
-2.4	-2.9	-3.0	-2.9																																
-2.9	-3.0	-2.9	-2.4																																
-3.0	-2.9	-2.4	0.0																																
$k = 1$	<table> <tr><td>0.0</td><td>-1.0</td><td>-1.0</td><td>-1.0</td></tr> <tr><td>-1.0</td><td>-1.0</td><td>-1.0</td><td>-1.0</td></tr> <tr><td>-1.0</td><td>-1.0</td><td>-1.0</td><td>-1.0</td></tr> <tr><td>-1.0</td><td>-1.0</td><td>-1.0</td><td>0.0</td></tr> </table>	0.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	0.0	$k = 10$	<table> <tr><td>0.0</td><td>-6.1</td><td>-8.4</td><td>-9.0</td></tr> <tr><td>-6.1</td><td>-7.7</td><td>-8.4</td><td>-8.4</td></tr> <tr><td>-8.4</td><td>-8.4</td><td>-7.7</td><td>-6.1</td></tr> <tr><td>-9.0</td><td>-8.4</td><td>-6.1</td><td>0.0</td></tr> </table>	0.0	-6.1	-8.4	-9.0	-6.1	-7.7	-8.4	-8.4	-8.4	-8.4	-7.7	-6.1	-9.0	-8.4	-6.1	0.0
0.0	-1.0	-1.0	-1.0																																
-1.0	-1.0	-1.0	-1.0																																
-1.0	-1.0	-1.0	-1.0																																
-1.0	-1.0	-1.0	0.0																																
0.0	-6.1	-8.4	-9.0																																
-6.1	-7.7	-8.4	-8.4																																
-8.4	-8.4	-7.7	-6.1																																
-9.0	-8.4	-6.1	0.0																																
$k = 2$	<table> <tr><td>0.0</td><td>-1.7</td><td>-2.0</td><td>-2.0</td></tr> <tr><td>-1.7</td><td>-2.0</td><td>-2.0</td><td>-2.0</td></tr> <tr><td>-2.0</td><td>-2.0</td><td>-2.0</td><td>-1.7</td></tr> <tr><td>-2.0</td><td>-2.0</td><td>-1.7</td><td>0.0</td></tr> </table>	0.0	-1.7	-2.0	-2.0	-1.7	-2.0	-2.0	-2.0	-2.0	-2.0	-2.0	-1.7	-2.0	-2.0	-1.7	0.0	$k = \infty$	<table> <tr><td>0.0</td><td>-14.</td><td>-20.</td><td>-22.</td></tr> <tr><td>-14.</td><td>-18.</td><td>-20.</td><td>-20.</td></tr> <tr><td>-20.</td><td>-20.</td><td>-18.</td><td>-14.</td></tr> <tr><td>-22.</td><td>-20.</td><td>-14.</td><td>0.0</td></tr> </table>	0.0	-14.	-20.	-22.	-14.	-18.	-20.	-20.	-20.	-20.	-18.	-14.	-22.	-20.	-14.	0.0
0.0	-1.7	-2.0	-2.0																																
-1.7	-2.0	-2.0	-2.0																																
-2.0	-2.0	-2.0	-1.7																																
-2.0	-2.0	-1.7	0.0																																
0.0	-14.	-20.	-22.																																
-14.	-18.	-20.	-20.																																
-20.	-20.	-18.	-14.																																
-22.	-20.	-14.	0.0																																

정책 평가: 격자 예 (3)

□ 각 단계마다 모든 상태에 대해 벨만 기대 방정식 적용

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')]$$

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

- k=1 단계

- 모든 비단말 상태에 대해

$$v_2 = 4 \times (0.25 \times (-1 + 0)) = -1$$

- 모든 4개의 방향의 행동이 동일한 보상 -1

k = 1

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

- k=2 단계

- 0행 1열의 상태에 대해

$$\begin{aligned} v_3 &= 1 \times (0.25 \times (-1 + 0)) + 3 \times (0.25 \times (-1 + -1)) \\ &= -1.75 \end{aligned}$$

k = 2

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

정책 평가 알고리즘

Iterative policy evaluation

Input π , the policy to be evaluated

Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$

Repeat

$$\Delta \leftarrow 0$$

For each $s \in \mathcal{S}$:

$$v \leftarrow V(s)$$

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until $\Delta < \theta$ (a small positive number)

Output $V \approx v_\pi$

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')]$$

정책 평가 알고리즘: 파이썬

```
import numpy as np
```

```
## 정책 평가 알고리즘
```

```
def policy_evaluation(pi, P, gamma=1.0, theta=1e-10): # 정책, 환경의 상태, 할인율, 세타(수렴 값)
```

```
    prev_V = np.zeros(len(P), dtype=np.float64) # 이전 가치 함수(k) 를 0으로 초기화
```

```
    # 수렴할 때까지 반복
```

```
    while True:
```

```
        V = np.zeros(len(P), dtype=np.float64) # 현재 가치 함수(k+1)를 0으로 초기화
```

```
        # 모든 상태들에 대해 반복
```

```
        for s in range(len(P)):
```

```
            # 현재 상태에서 가능한 전이에 대해 반복
```

```
            for prob, next_state, reward, done in P[s][pi(s)]:
```

```
                V[s] += prob * (reward + gamma * prev_V[next_state] * (not done)) # 종료 상태(done)인 경우엔 0
```

```
            if np.max(np.abs(prev_V - V)) < theta: # 수렴한 경우 루프 빠져나옴
```

```
                break
```

```
            prev_V = V.copy() # 현재 상태를 이전 상태로 복사
```

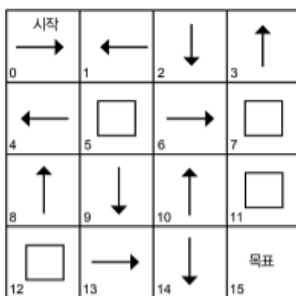
```
    return V # 계산된 가치함수 값을 리턴
```

```
14: {0: [(0.3333333333333333, 10, 0.0, False),
      (0.3333333333333333, 13, 0.0, False),
      (0.3333333333333333, 14, 0.0, False)],
  1: [(0.3333333333333333, 13, 0.0, False),
      (0.3333333333333333, 14, 0.0, False),
      (0.3333333333333333, 15, 1.0, True)],
  2: [(0.3333333333333333, 14, 0.0, False),
      (0.3333333333333333, 15, 1.0, True),
      (0.3333333333333333, 10, 0.0, False)],
  3: [(0.3333333333333333, 15, 1.0, True),
      (0.3333333333333333, 10, 0.0, False),
      (0.3333333333333333, 13, 0.0, False)],
  15: {0: [(1.0, 15, 0, True)],
       1: [(1.0, 15, 0, True)],
       2: [(1.0, 15, 0, True)],
       3: [(1.0, 15, 0, True)]}}
```

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) [r + \gamma v_k(s')]$$

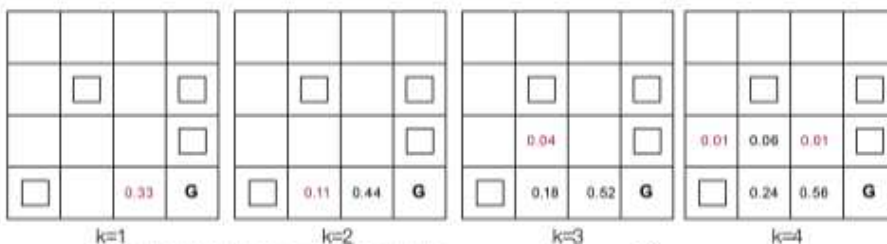
4. 동적 계획법

프로즌 레이크 정책 평가 예 - 임의의 정책

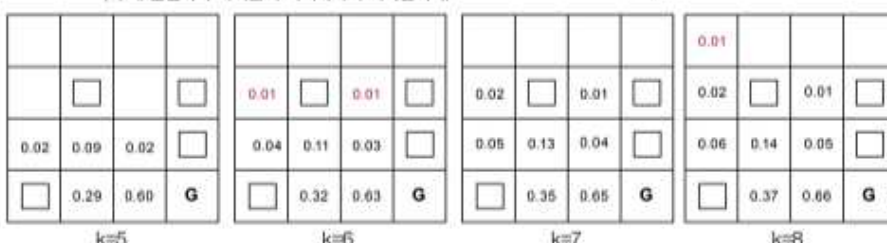


시작	←	↓	↑
0.0955	0.0471	0.0470	0.0456
←	□	→	□
0.1469		0.0498	
↑	↓	↑	□
0.2028	0.2647	0.1038	
□	→	↓	목표
	0.4957	0.7417	

(1) 218번의 순환 이후에, 정책 평가 알고리즘은 이런 값으로 수렴합니다 (여기서는 수렴에 대한 임계값으로 10^{-6} 을 사용했습니다).



(1) 매 순환마다 가치들이 퍼져나가기 시작합니다.



(2) 가치가 퍼져나가면서 점점 정확해집니다.

프로즌 레이크 정책 평가 코드

- ❑ 프로즌 레이크의 임의의 정책에 대해 정책 평가 파이썬 알고리즘을 적용한 예
- ❑ 출력 지원 모듈 outputHelper 사용
 - 정책 출력: `print_policy()`
 - 적용된 정책 출력
 - 정책 성공 확률 출력: `probability_success()`
 - 100개의 임의의 에피소드에 대해 적용
 - 한 에피소드 당 최대 200번 스텝 적용
 - 평균 리턴 값 출력: `mean_return()`
 - 100개의 임의의 에피소드에 대해 적용
 - 한 에피소드 당 최대 200번 스텝 적용
- ❑ 코드 출처
 - <https://goodboychan.github.io/book/>

프로즌 레이크 예: 정책 평가 코드 (1)

```
## 정책 평가 알고리즘
def policy_evaluation(pi, P, gamma=1.0, theta=1e-10): # 정책, 환경의 상태, 할인율, 세타(수렴 값)
    prev_V = np.zeros(len(P), dtype=np.float64) # 이전 가치 함수(k) 를 0으로 초기화
    # 수렴할 때까지 반복
    while True:
        V = np.zeros(len(P), dtype=np.float64) # 현재 가치 함수(k+1)를 0으로 초기화
        # 모든 상태들에 대해 반복
        for s in range(len(P)):
            # 현재 상태에서 가능한 전이에 대해 반복
            for prob, next_state, reward, done in P[s][pi(s)]:
                V[s] += prob * (reward + gamma * prev_V[next_state] * (not done)) # 종료 상태(done)인 경우
        엔 0
        if np.max(np.abs(prev_V - V)) < theta: # 수렴한 경우 루프 빠져나옴
            break
        prev_V = V.copy() # 현재 상태를 이전 상태로 복사
    return V # 계산된 가치함수 값을 리턴
```


프로즌 레이크 예: 정책 평가 코드 (2)

```
import gym
from outputHelper import *

## 프로즌 레이크 환경 생성
env = gym.make('FrozenLake-v1')
# 상태 저장
P = env.env.P

init_state = env.reset(seed=123) # 환경 초기화
goal_state = 15 # 목표 상태

## 임의의 정책 생성
LEFT, DOWN, RIGHT, UP = range(4)

random_pi = lambda s: {
    0:RIGHT, 1:LEFT, 2:DOWN, 3:UP,
    4:LEFT, 5:LEFT, 6:RIGHT, 7:LEFT,
    8:UP, 9:DOWN, 10:UP, 11:LEFT,
    12:LEFT, 13:RIGHT, 14:DOWN, 15:LEFT
}[s]
```

```
# 정책, 성공 확률, 평균 리턴 값 출력
print_policy(random_pi, P)
print('성공확률 {:.2f}%. 평균리턴값 {:.4f}'.format(
    probability_success(env, random_pi,
        goal_state=goal_state)*100,
    mean_return(env, random_pi)))

## 정책 평가 수행
V = policy_evaluation(random_pi, P, gamma=0.99)

# 상태-가치 함수 출력
print_state_value_function(V, P, prec=4)
```

프로즌 레이크 예: outputHelper 모듈 (1)

```
import numpy as np
import random

## 정책을 출력
def print_policy(pi, P, action_symbols=('<', 'v', '>', '^'), n_cols=4, title='정책:'):
    print(title)
    arrs = {k:v for k,v in enumerate(action_symbols)}
    for s in range(len(P)):
        a = pi(s)
        print("| ", end="")
        if np.all([done for action in P[s].values() for _, _, done in action]):
            print("".rjust(9), end=" ")
        else:
            print(str(s).zfill(2), arrs[a].rjust(6), end=" ")
        if (s + 1) % n_cols == 0: print("")

## 정책 성공 확률 출력
def probability_success(env, pi, goal_state, n_episodes=100, max_steps=200):
    random.seed(123); np.random.seed(123)
    results = []
    for _ in range(n_episodes):
        s, done, steps = env.reset(seed=123), False, 0
        state = s[0] # s => (0, {'prob': 1})
        while not done and steps < max_steps:
            state, _, done, _ = env.step(pi(state))
            steps += 1
        results.append(state == goal_state)
    return np.sum(results)/len(results)
```

```
14: {0: [(0.3333333333333333, 10, 0.0, False),
        (0.3333333333333333, 13, 0.0, False),
        (0.3333333333333333, 14, 0.0, False)],
    1: [(0.3333333333333333, 13, 0.0, False),
        (0.3333333333333333, 14, 0.0, False),
        (0.3333333333333333, 15, 1.0, True)],
    2: [(0.3333333333333333, 14, 0.0, False),
        (0.3333333333333333, 15, 1.0, True),
        (0.3333333333333333, 10, 0.0, False)],
    3: [(0.3333333333333333, 15, 1.0, True),
        (0.3333333333333333, 10, 0.0, False),
        (0.3333333333333333, 13, 0.0, False)],
    15: {0: [(1.0, 15, 0, True)],
        1: [(1.0, 15, 0, True)],
        2: [(1.0, 15, 0, True)],
        3: [(1.0, 15, 0, True)]}}
```

프로즌 레이크 예: outputHelper 모듈

```

## 평균 리턴 값 출력
def mean_return(env, pi, n_episodes=100, max_steps=200):
    random.seed(123); np.random.seed(123)
    results = []
    for _ in range(n_episodes):
        s, done, steps = env.reset(seed=123), False, 0
        state = s[0]          # s => (0, {'prob': 1})
        results.append(0.0)
        while not done and steps < max_steps:
            state, reward, done, _, _ = env.step(pi(state))
            results[-1] += reward
            steps += 1
        return np.mean(results)

## 상태 가치 함수 출력
def print_state_value_function(V, P, n_cols=4, prec=3, title='상태-가치 함수:'):
    print(title)
    for s in range(len(P)):
        v = V[s]
        print("|", end="")
        if np.all([done for action in P[s].values() for _, _, _, done in action]):
            print("".rjust(9), end=" ")
        else:
            print(str(s).zfill(2), '{}'.format(np.round(v, prec)).rjust(6), end=" ")
        if (s + 1) % n_cols == 0: print("|")

```

19

프로즌 레이크 예: 정책 평가 실행 예 (1)

```

1 import gym
2 from outputHelper import *
3
4 ## 프로즌 레이크 환경 생성
5 env = gym.make('FrozenLake-v1')
6 env.reset(seed=123)
7
8 # 상태 저장
9 P = env.env.P
10 P

```

```

{0: {0: [(0.3333333333333333, 0, 0.0, False),
(0.3333333333333333, 0, 0.0, False),
(0.3333333333333333, 4, 0.0, False)],
1: [(0.3333333333333333, 0, 0.0, False),
(0.3333333333333333, 4, 0.0, False),
(0.3333333333333333, 1, 0.0, False)],
2: [(0.3333333333333333, 4, 0.0, False),
(0.3333333333333333, 1, 0.0, False),
(0.3333333333333333, 0, 0.0, False)],
3: [(0.3333333333333333, 1, 0.0, False),
(0.3333333333333333, 0, 0.0, False),
(0.3333333333333333, 0, 0.0, False)]},
1: {0: [(0.3333333333333333, 1, 0.0, False),
(0.3333333333333333, 0, 0.0, False),

```

```

1 init_state = env.reset(seed=123) # 환경 초기화
2 goal_state = 15 # 목표 상태

```

```

1 ## 임의의 정책 생성
2 LEFT, DOWN, RIGHT, UP = range(4)
3 random_pi = lambda s: {
4     0:RIGHT, 1:LEFT, 2:DOWN, 3:UP,
5     4:LEFT, 5:LEFT, 6:RIGHT, 7:LEFT,
6     8:UP, 9:DOWN, 10:UP, 11:LEFT,
7     12:LEFT, 13:RIGHT, 14:DOWN, 15:LEFT
8 }[s]

```

```

1 # 정책, 성공 확률, 평균 리턴 값 출력
2 print_policy(random_pi, P)
3 print('성공확률 {:.2f}%, 평균리턴값 {:.4f}'.format(probability_success(env, random_pi, goal_state=goal_state)*100,
4     mean_return(env, random_pi)))

```

```

정책:
| 00      > | 01      < | 02      v | 03      ^ |
| 04      < |      | 06      > |      |
| 08      ^ | 09      v | 10      ^ |      |
|      | 13      > | 14      v |      |
성공확률 0.00%. 평균리턴값 0.0000.

```

20

프로즌 레이크 예: 정책 평가 실행 예 (2)

```
## 정책 평가 수행
V = policy_evaluation(random_pi, P, gamma=0.99)

# 상태-가치 함수 출력
print_state_value_function(V, P, prec=4)
```

상태-가치 함수:

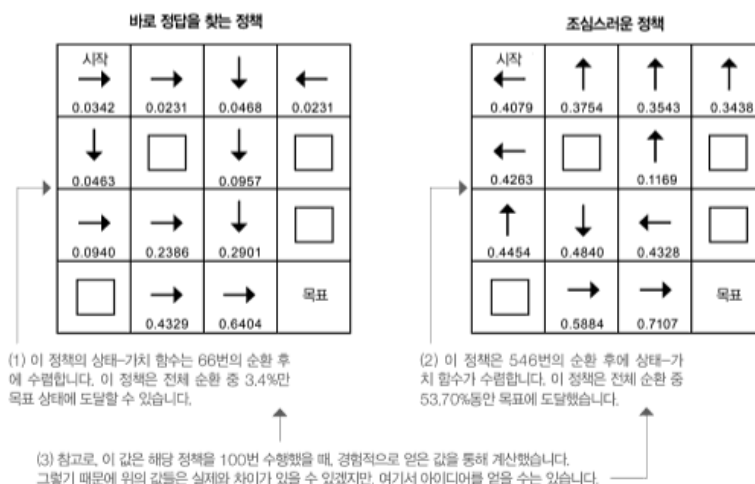
00 0.0955	01 0.0471	02 0.047	03 0.0456
04 0.1469		06 0.0498	
08 0.2028	09 0.2647	10 0.1038	
	13 0.4957	14 0.7417	

시작 →	←	↓	↑
0.0955	0.0471	0.0470	0.0456
←	□	→	□
0.1469		0.0498	
↑	↓	↑	□
0.2028	0.2647	0.1038	
□	→	↓	목표
	0.4957	0.7417	

(1) 218번의 순환 이후에, 정책 평가 알고리즘은 이런 값으로 수렴합니다 (여기서는 수렴에 대한 임계값으로 10^{-10} 을 사용했습니다).

과제 4-1: 정책 평가

- 앞의 프로즌 레이크 예의 임의의 정책 평가 실행
- 아래의 정책에 대해 각각 적용하고 실행 분석
 - 바로 정답을 찾는 정책
 - 조심스러운 정책



정책 개선 (Policy Improvement)

4. 동적 계획법

정책 개선 (Policy Improvement)

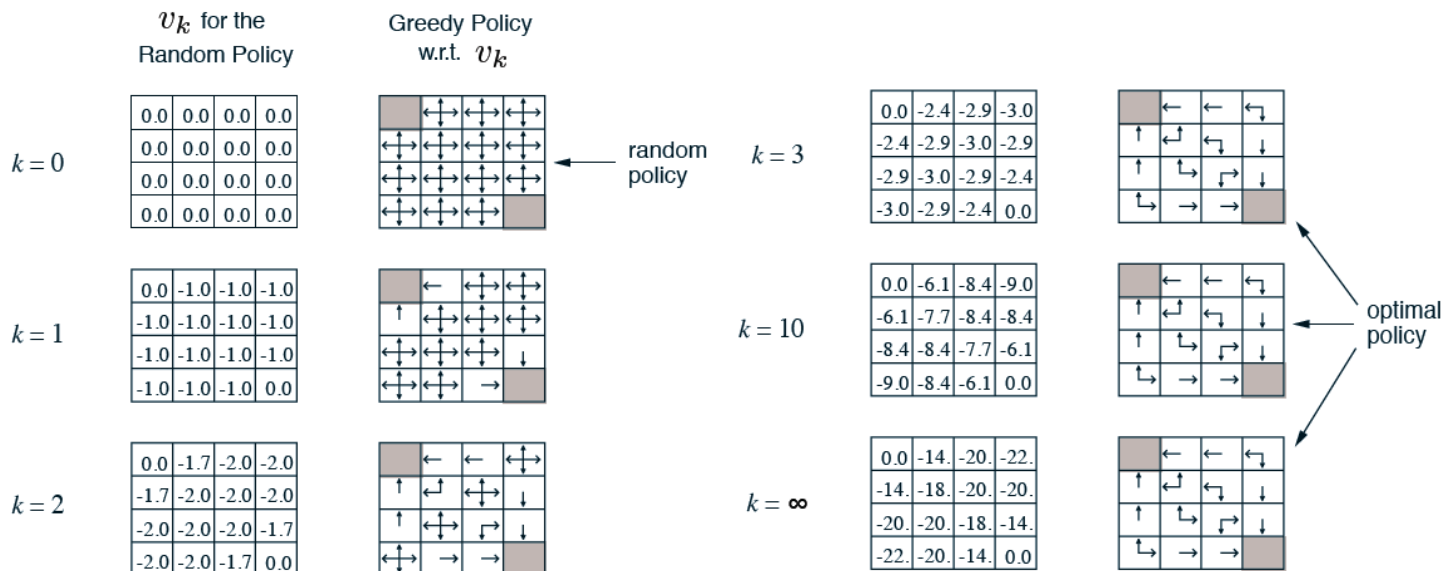
- 주어진 정책에 대해 올바른 가치함수를 계산한 후 더 나은 정책으로 개선
- 정책 개선 중 그리디 개선 (Greedy Improvement, 탐욕 개선)은 다음 상태 중 가장 높은 가치함수 값을 가진 상태로의 이동을 선택

$$\pi' = \text{greedy}(v_{\pi})$$

$$\begin{aligned}\pi'(s) &= \operatorname{argmax}_a \sum_{s', r} p(s', r | s, a) \left[r + \gamma v_{\pi}(s') \right] \\ &= \operatorname{argmax}_a q_{\pi}(s, a)\end{aligned}$$

□ 각 반복에 대해 정책 개선

- 격자 세상 예에서는 3번의 정책 개선으로 최적의 정책을 도출



정책 개선 알고리즘: 파이썬

정책 개선 알고리즘

```
def policy_improvement(V, P, gamma=1.0):    # 상태-가치함수, 환경의 상태, 할인율 인수
    Q = np.zeros((len(P), len(P[0])), dtype=np.float64)    # 상태-가치 함수를 0으로 초기화
    # 모든 상태들에 반복
    for s in range(len(P)):
        # 현재 상태에서 모든 가능한 행동에 대해 반복
        for a in range(len(P[s])):
            # 현재 상태, 행동에서 가능한 전이에 대해 반복
            for prob, next_state, reward, done in P[s][a]:
                Q[s][a] += prob * (reward + gamma * V[next_state] * (not done))    # 종료 상태(done)인 경우엔 0
    # 그리디 개선으로 정책을 개선
    new_pi = lambda s: {s:a for s, a in enumerate(np.argmax(Q, axis=1))}[s]
    return new_pi    # 개선된 새로운 정책을 리턴
```

$$\begin{aligned}\pi'(s) &= \operatorname{argmax}_a \sum_{s', r} p(s', r | s, a) \left[r + \gamma v_{\pi}(s') \right] \\ &= \operatorname{argmax}_a q_{\pi}(s, a)\end{aligned}$$

프로즌 레이크 정책 개선 예 - 임의의 정책

시작 0 →	1 ←	2 ↓	3 ↑
4 ←	5 □	6 →	7 □
8 ↑	9 ↓	10 ↑	11 □
12 □	13 →	14 ↓	목표 15

시작 0.0955 →	0.0471 ←	0.0470 ↓	0.0456 ↑
0.1469 ←	□	0.0498 →	□
0.2028 ↑	0.2647 ↓	0.1038 ↑	□
□	0.4957 →	0.7417 ↓	목표

(1) 218번의 순환 이후에, 정책 평가 알고리즘은 이런 값으로 수렴합니다 (여기서는 수렴에 대한 임계값으로 10^{-6} 을 사용했습니다).

k=1	k=2	k=3	k=4
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □
0.33 G	0.11 0.44 G	0.04 0.18 0.52 G	0.01 0.06 0.01 0.24 0.56 G

(1) 매 순환마다 가치들이 퍼져나가기 시작합니다.

k=5	k=6	k=7	k=8
□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □	□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □	0.01 □ □ □ □ 0.02 □ □ □ □ 0.06 0.14 0.05 □ □ □ □ □ □
0.02 0.09 0.02 □ □ 0.29 0.60 G	0.01 □ 0.01 □ 0.04 0.11 0.03 □ □ 0.32 0.63 G	0.02 □ 0.01 □ 0.05 0.13 0.04 □ □ 0.35 0.65 G	0.02 □ 0.01 □ 0.06 0.14 0.05 □ □ 0.37 0.66 G

(2) 가치가 퍼져나감에 따라 점점 정확해집니다.

:학과

27

프로즌 레이크 예: 정책 개선

정책 개선 수행

improved_pi = policy_improvement(V, P)

개선된 정책, 성공 확률, 평균 리턴 값 출력

print_policy(improved_pi, P)

print('성공확률 {:.2f}%. 평균리턴값 {:.4f}.'.format(

probability_success(env, improved_pi, goal_state=goal_state)*100, mean_return(env, improved_pi)))

```
1 ## 정책 개선 수행
2 improved_pi = policy_improvement(V, P)
3
4 # 개선된 정책, 성공 확률, 평균 리턴 값 출력
5 print_policy(improved_pi, P)
6 print('성공확률 {:.2f}%. 평균리턴값 {:.4f}.'.format(
7     probability_success(env, improved_pi, goal_state=goal_state)*100,
8     mean_return(env, improved_pi)))
```

정책:

00	<	01	^	02	^	03	^
04	<			06	<		
08	^	09	v	10	<		
		13	>	14	v		

성공확률 100.00%, 평균리턴값 1.0000.

프로즌 레이크 개선 코드

❑ 프로즌 레이크의 임의의 정책에 대해 정책 개선

- 앞의 정책 평가 수행 후의 상태-가치 함수를 적용하여 정책 개선
- 정책 개선 전 0%의 성공 확률이 개선 후에 100%로 향상

정책:

00	>	01	<	02	v	03	^
04	<			06	>		
08	^	09	v	10	^		
		13	>	14	v		

성공확률 0.00%, 평균리턴값 0.0000.

정책 평가 후

상태-가치 함수:

00	0.0955	01	0.0471	02	0.047	03	0.0456
04	0.1469			06	0.0498		
08	0.2028	09	0.2647	10	0.1038		
		13	0.4957	14	0.7417		

정책 개선 후

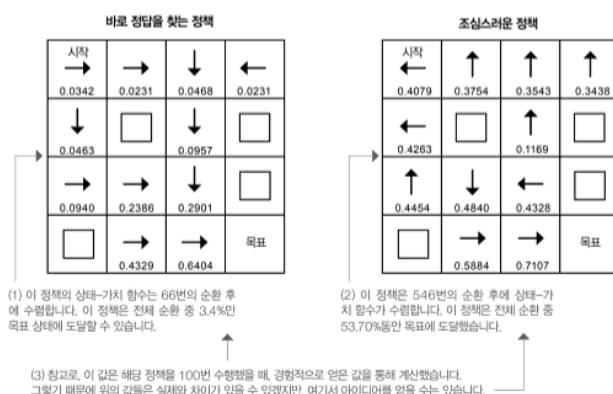
정책:

00	<	01	^	02	^	03	^
04	<			06	<		
08	^	09	v	10	<		
		13	>	14	v		

성공확률 100.00%, 평균리턴값 1.0000.

과제 4-2: 정책 개선

- ❑ 앞의 프로즌 레이크 예의 임의의 정책에 대해 정책 개선 실행
- ❑ 아래의 정책에 대해 각각 적용하고 실행 분석
 - 바로 정답을 찾는 정책
 - 조심스러운 정책

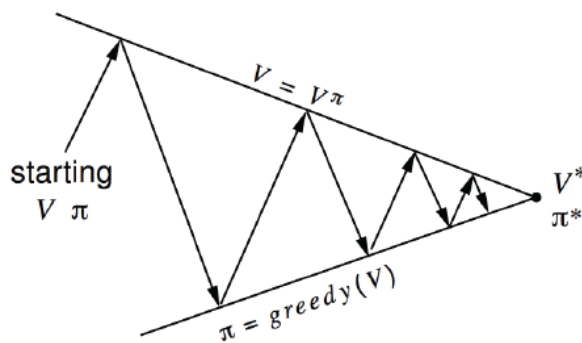


정책 반복 (Policy Iteration)

4. 동적 계획법

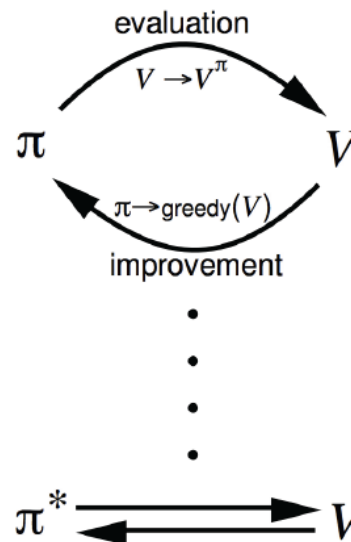
정책 반복(Policy Iteration): 일반화된 정책 반복

- 격자 세상과 같은 작은 상태에 대해서는 한 번의 개선으로 최적의 정책을 구하지만 일반적으로는 **정책 평가와 정책 개선을 반복**하여 최적의 정책을 구함
 - 일반화된 정책 반복(Generalized Policy Iteration, GPI)라고 함



Policy evaluation Estimate v_π
Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$
Greedy policy improvement



Policy iteration (using iterative policy evaluation)

1. Initialization

$V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation

Repeat

$\Delta \leftarrow 0$

For each $s \in \mathcal{S}$:

$v \leftarrow V(s)$

$V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s)) [r + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$ (a small positive number)

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')]$$

3. Policy Improvement

policy-stable \leftarrow true

For each $s \in \mathcal{S}$:

old-action $\leftarrow \pi(s)$

$\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$

If *old-action* $\neq \pi(s)$, then *policy-stable* \leftarrow false

If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

$$\begin{aligned} \pi'(s) &= \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a) [r + \gamma v_\pi(s')] \\ &= \operatorname{argmax}_a q_\pi(s,a) \end{aligned}$$

프로즌 레이크 예: 정책 반복 코드 (1)

```
## 정책 반복 수행
```

```
# 적대적인 정책 생성
```

```
adversarial_pi = lambda s: {
    0:UP, 1:UP, 2:UP, 3:UP,
    4:UP, 5:LEFT, 6:UP, 7:LEFT,
    8:LEFT, 9:LEFT, 10:LEFT, 11:LEFT,
    12:LEFT, 13:LEFT, 14:LEFT, 15:LEFT
}[s]
```

```
print_policy(adversarial_pi, P)
```

```
print('성공확률 {:.2f}%. 평균리턴값 {:.4f}'.format(
    probability_success(env, adversarial_pi, goal_state=goal_state)*100,
    mean_return(env, adversarial_pi)))
```

```
V = np.zeros(len(P), dtype=np.float64) # 가치 함수를 0으로 초기화
```

```
n = 0
```

```
pi = adversarial_pi
```

더 이상 정책 개선이 안될 때까지 반복

while True:

old_pi = {s:pi(s) for s in range(len(P))} # 이전 반복의 정책

정책 평가 수행

V = policy_evaluation(pi, P, gamma=0.99)

정책 개선 수행

pi = policy_improvement(V, P, gamma=0.99)

정책, 상태-가치 함수, 성공 확률, 평균 리턴 값 출력

print("----- 반복 {:d}".format(n))

print_policy(pi, P) # 정책

print_state_value_function(V, P, prec=4) # 상태-가치 함수

print('성공확률 {:.2f}%. 평균리턴값 {:.4f}'.format(

probability_success(env, pi, goal_state=goal_state)*100,

mean_return(env, pi)))

n=n+1

더 이상 정책 개선이 되지 않으면 반복 중단

if old_pi == {s:pi(s) for s in range(len(P))}:

break

4. 동적 계획법

프로즌 레이크 예: 정책 반복
실행 예

```

22 # 더 이상 정책 개선이 되지 않으면 반복 중단
23 if old_pi == {s:pi(s) for s in range(len(P))}:
24     break

```

----- 반복 0

```

정책:
| 00 < | 01 < | 02 < | 03 < |
| 04 < | 05 < | 06 < | 07 < |
| 08 < | 09 < | 10 < | 11 < |
| 12 < | 13 < | 14 v | 15 < |

```

```

상태-가치 함수:
| 00 0.0 | 01 0.0 | 02 0.0 | 03 0.0 |
| 04 0.0 | 05 0.0 | 06 0.0 | 07 0.0 |
| 08 0.0 | 09 0.0 | 10 0.0 | 11 0.0 |
| 12 0.0 | 13 0.0 | 14 0.0 | 15 0.0 |

```

성공확률 0.00%. 평균리턴값 0.0000.

----- 반복 1

```

정책:
| 00 < | 01 v | 02 > | 03 ^ |
| 04 < | 05 < | 06 < | 07 < |
| 08 < | 09 v | 10 < | 11 < |
| 12 < | 13 v | 14 > | 15 < |

```

```

상태-가치 함수:
| 00 0.0 | 01 0.0 | 02 0.0366 | 03 0.018 |
| 04 0.0 | 05 0.0 | 06 0.0744 | 07 0.0 |
| 08 0.0 | 09 0.0 | 10 0.1887 | 11 0.0 |
| 12 0.0 | 13 0.0 | 14 0.4975 | 15 0.0 |

```

성공확률 0.00%. 평균리턴값 0.0000.

----- 반복 2

```

정책:
| 00 v | 01 > | 02 > | 03 ^ |
| 04 < | 05 < | 06 < | 07 < |
| 08 v | 09 v | 10 < | 11 < |
| 12 < | 13 > | 14 > | 15 < |

```

```

상태-가치 함수:
| 00 0.0 | 01 0.0524 | 02 0.1587 | 03 0.154 |
| 04 0.0 | 05 0.0 | 06 0.1681 | 07 0.0 |
| 08 0.0 | 09 0.2247 | 10 0.3509 | 11 0.0 |
| 12 0.0 | 13 0.3302 | 14 0.6703 | 15 0.0 |

```

성공확률 0.00%. 평균리턴값 0.0000.

----- 반복 3

```

정책:
| 00 < | 01 ^ | 02 > | 03 ^ |
| 04 < | 05 < | 06 < | 07 < |
| 08 ^ | 09 v | 10 < | 11 < |
| 12 < | 13 > | 14 v | 15 < |

```

```

상태-가치 함수:
| 00 0.1213 | 01 0.0948 | 02 0.1924 | 03 0.1867 |
| 04 0.1515 | 05 0.2039 | 06 0.2039 | 07 0.0 |
| 08 0.1863 | 09 0.3783 | 10 0.4255 | 11 0.0 |
| 12 0.0 | 13 0.5346 | 14 0.7071 | 15 0.0 |

```

성공확률 100.00%. 평균리턴값 1.0000.

----- 반복 4

```

정책:
| 00 < | 01 ^ | 02 < | 03 ^ |
| 04 < | 05 < | 06 < | 07 < |
| 08 ^ | 09 v | 10 < | 11 < |
| 12 < | 13 > | 14 v | 15 < |

```

```

상태-가치 함수:
| 00 0.5198 | 01 0.3844 | 02 0.2608 | 03 0.2531 |
| 04 0.5355 | 05 0.2763 | 06 0.2763 | 07 0.0 |
| 08 0.5675 | 09 0.6167 | 10 0.5766 | 11 0.0 |
| 12 0.0 | 13 0.7245 | 14 0.8544 | 15 0.0 |

```

성공확률 100.00%. 평균리턴값 1.0000.

----- 반복 5

```

정책:
| 00 < | 01 ^ | 02 ^ | 03 ^ |
| 04 < | 05 < | 06 < | 07 < |
| 08 ^ | 09 v | 10 < | 11 < |
| 12 < | 13 > | 14 v | 15 < |

```

```

상태-가치 함수:
| 00 0.5325 | 01 0.4498 | 02 0.3807 | 03 0.3695 |
| 04 0.5486 | 05 0.3232 | 06 0.3232 | 07 0.0 |
| 08 0.5814 | 09 0.6318 | 10 0.5987 | 11 0.0 |
| 12 0.0 | 13 0.7344 | 14 0.8592 | 15 0.0 |

```

성공확률 100.00%. 평균리턴값 1.0000.

----- 반복 6

```

정책:
| 00 < | 01 ^ | 02 ^ | 03 ^ |
| 04 < | 05 < | 06 < | 07 < |
| 08 ^ | 09 v | 10 < | 11 < |
| 12 < | 13 > | 14 v | 15 < |

```

```

상태-가치 함수:
| 00 0.542 | 01 0.4988 | 02 0.4707 | 03 0.4569 |
| 04 0.5585 | 05 0.3583 | 06 0.3583 | 07 0.0 |
| 08 0.5918 | 09 0.6431 | 10 0.6152 | 11 0.0 |
| 12 0.0 | 13 0.7417 | 14 0.8628 | 15 0.0 |

```

성공확률 100.00%. 평균리턴값 1.0000.

과제 4-3: 정책 반복

- 앞의 프로즌 레이크의 적대적인 정책에 대해 실행
 - 아래의 정책에 대해서도 각각 적용하고 실행 분석
 - 바로 정답을 찾는 정책
 - 조심스러운 정책

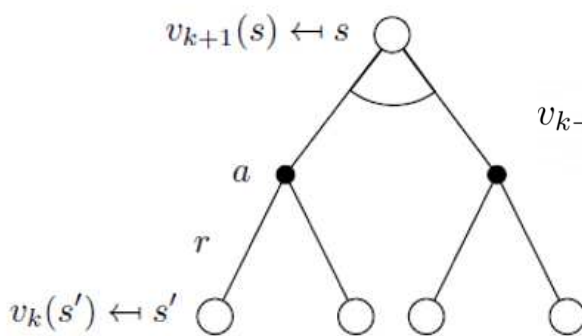
가치 반복 (Value Iteration)

가치 반복 (Value Iteration) (1)

□ 벨만 최적화 방정식 (Bellman Optimality Equation)을 사용하여 현재 주어진 정책에 대해 가치함수를 계산

- 가치함수 계산 시 최대값이 되는 행동 만을 선택
- 정책 반복에 비해 계산 감소
- 최대값 선택이 그리디 개선 효과를 가져옴

$$v_*(s) = \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_*(s')] \quad q_\pi(s, a)$$

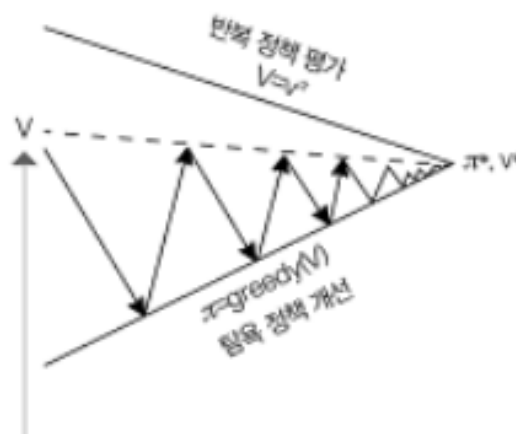


$$v_{k+1}(s) = \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_k(s')]$$

$q(s, a)$

가치 반복 (2)

가치 반복



(2) 가치 반복은 임의의 가치 함수를 가지고 시작하여, 정책 평가의 일부 과정을 수행합니다.

가치 반복 알고리즘

Value iteration

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat

$\Delta \leftarrow 0$

For each $s \in \mathcal{S}$:

$v \leftarrow V(s)$

$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$ (a small positive number)

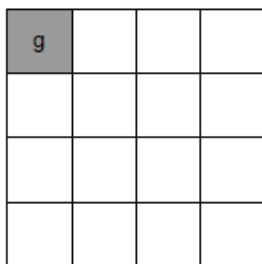
Output a deterministic policy, $\pi \approx \pi_*$, such that

$\pi(s) = \arg\max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')]$$

$q(s,a)$

격자 예: 가치 반복(1)



Problem

- ❑ 1개의 단말 상태 (terminal state)
- ❑ 상하 좌우 4개의 이동 가능한 행동
- ❑ 모든 이동에 대해 할인율 1
- ❑ 비단말 상태 이동 시 보상은 -1

격자 예: 가치 반복 (2)

g			

Problem

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

V_1

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

V_2

0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-2
-2	-2	-2	-2

V_3

0	-1	-2	-3
-1	-2	-3	-3
-2	-3	-3	-3
-3	-3	-3	-3

V_4

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-4
-3	-4	-4	-4

V_5

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-5

V_6

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

V_7

격자 예: 가치 반복 (3)

□ 각 단계마다 모든 상태에 대해 벨만 방정식 적용

$$v_{k+1}(s) = \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_k(s')]$$

- k=3 단계

- 0행 1열의 상태

$$v_3 = \max ((-1+0), (-1+-1), (-1+-1), (-1+-1)) = -1$$

- 0행 2열의 상태

$$v_3 = \max ((-1+-1), (-1+-1), (-1+-1), (-1+-1)) = -2$$

- k=4 단계

- 0행 1열의 상태

$$v_3 = \max ((-1+0), (-1+-2), (-1+-2), (-1+-2)) = -1$$

- 0행 2열의 상태

$$v_3 = \max ((-1+-1), (-1+-2), (-1+-2), (-1+-2)) = -2$$

- 0행 3열의 상태

$$v_3 = \max ((-1+-2), (-1+-2), (-1+-2), (-1+-2)) = -3$$

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

 V_2

0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-2
-2	-2	-2	-2

 V_3

0	-1	-2	-3
-1	-2	-3	-3
-2	-3	-3	-3
-3	-3	-3	-3

 V_4


가치 반복 알고리즘

```
def value_iteration(P, gamma=1.0, theta=1e-10):    # 정책, 환경의 상태, 할인율, 세타(수렴 값)
    V = np.zeros(len(P), dtype=np.float64)
    # 수렴할 때까지 반복
    while True:
        Q = np.zeros((len(P), len(P[0])), dtype=np.float64)    # Q-함수를 0으로 초기화
        # 모든 상태들에 대해 반복
        for s in range(len(P)):
            # 현재 상태, 행동에서 가능한 전이에 대해 반복
            for a in range(len(P[s])):
                for prob, next_state, reward, done in P[s][a]:
                    Q[s][a] += prob * (reward + gamma * V[next_state] * (not done))
            # 최대 Q-함수(그리디 정책에 의한 새로운 가치와 이전 가치와의 차이가 세타보다 적으면(수렴하면) 종료
            if np.max(np.abs(V - np.max(Q, axis=1))) < theta:
                break
        V = np.max(Q, axis=1)    # 새로운 가치 함수 갱신

    # 최종 정책(최적화된 정책)
    pi = lambda s: {s:a for s, a in enumerate(np.argmax(Q, axis=1))}[s]

    return V, pi    # 가치 함수와 정책 리턴
```

$$v_{k+1}(s) = \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_k(s')]$$


 $q(s, a)$

프로즌 레이크 예: 가치 반복 코드

가치 반복 수행

```
V_best_v, pi_best_v = value_iteration(P, gamma=0.99)

print('최적화된 정책과 상태-가치 함수 (VI):')
print_policy(pi_best_v, P)
print('성공확률 {:.2f}%. 평균리턴값 {:.4f}'.format(
    probability_success(env, pi_best_v, goal_state=goal_state)*100,
    mean_return(env, pi_best_v)))

print()
print_state_value_function(V_best_v, P, prec=4)
```

프로즌 레이크 예: 가치 반복 실행 예

```

1  ## 가치 반복 수행
2  V_best_v, pi_best_v = value_iteration(P, gamma=0.99)
3
4  print('최적화된 정책과 상태-가치 함수 (VI):')
5  print_policy(pi_best_v, P)
6  print('성공확률 {:.2f}%, 평균리턴값 {:.4f}'.format(
7      probability_success(env, pi_best_v, goal_state=goal_state)*100,
8      mean_return(env, pi_best_v)))
9  print()
10 print_state_value_function(V_best_v, P, prec=4)

```

최적화된 정책과 상태-가치 함수 (VI):

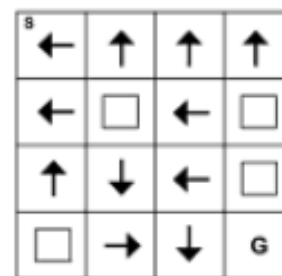
정책:

00	<	01	^	02	^	03	^
04	<			06	<		
08	^	09	v	10	<		
		13	>	14	v		

성공확률 100.00%, 평균리턴값 1.0000.

상태-가치 함수:

00	0.542	01	0.4988	02	0.4707	03	0.4569
04	0.5585			06	0.3583		
08	0.5918	09	0.6431	10	0.6152		
		13	0.7417	14	0.8628		



과제 4-4: 가치 반복

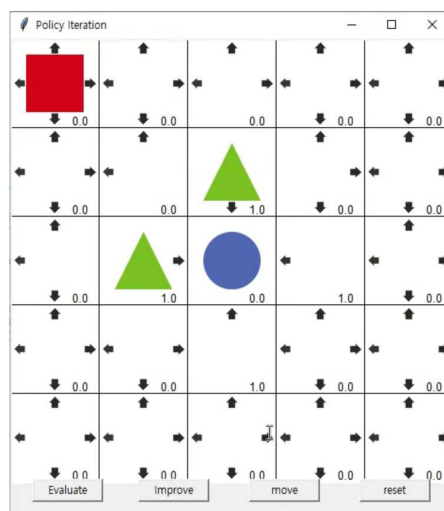
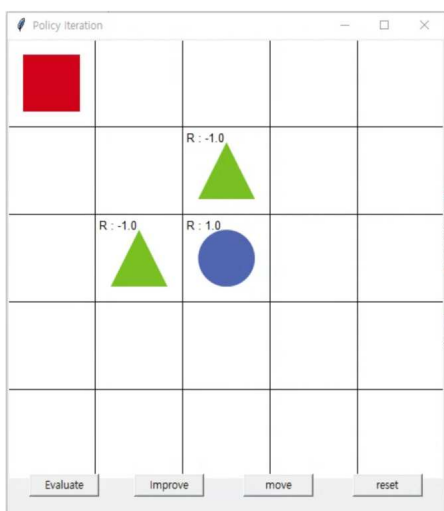
□ 앞의 프로즌 레이크의 가치 반복에 대해 실행

그리드 월드 (Grid World) 예

4. 동적 계획법

그리드 월드(Grid World) 예: 정책 반복 (1)

- 5 x 5 그리드 상에서 에이전트(빨간색 사각형)가 목표 지점(파란색 원)로 이동하는 문제
 - 버튼: 정책 평가, 정책 개선, 이동, 초기화
 - 반복하지 않고 버튼을 누를때 마다 한번씩 정책을 평가/개선
 - 코드 출처
 - <https://github.com/rlcode/reinforcement-learning-kr-v2>



그리드 월드(Grid World) 예: 정책 반복 (2)

□ 그리드 환경, `GridPI_environment.py`

- 보상
 - 장애물(녹색 삼각형)에 이동 시 -1 보상
 - 목표 지점 도착 시 +1 보상
- 환경의 상태: 그리드 위치 -> [열(x), 행(y)]
 - 가치함수 테이블: `value_table`, (5,5)
- 행동
 - 각 상태에서의 행동: 좌, 우, 상, 하 -> [0, 1, 2, 3]
- 정책
 - 각 상태에서의 행동의 확률: 좌/우/상/하 -> [0.25, 0.25, 0.25, 0.25]
 - 정책 테이블: `policy_table`, (4,5,5)
 - 결정론적 환경
 - 에이전트가 특정 상태에서 특정 행동을 취할 때 전이될 상태의 확률은 1, 나머지 행동의 전이 확률은 0

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) [r + \gamma v_k(s')]$$

파일 구성 및 메인: 정책 반복

□ 2개의 파일

- `GridPI.py`
 - 정책 반복 알고리즘 관련 함수(PolicyIteration 클래스), 메인
- `GridPI_environment.py`
 - 그리드월드 GUI 구성(GraphicDisplay 클래스)
 - 상태, 보상 등 환경 정보 제공 함수(Env 클래스)

```
import numpy as np
from GridPI_environment import GraphicDisplay, Env
.....

if __name__ == "__main__":
    env = Env()          # 그리드 환경 객체 생성
    policy_iteration = PolicyIteration(env)  # 정책 반복 객체 생성
    # GUI로 그리드 환경 디스플레이
    grid_world = GraphicDisplay(policy_iteration)
    grid_world.mainloop()
```

PolicyIteration 클래스

정책 반복

class PolicyIteration:

생성자: 가치함수 테이블, 정책 테이블 초기화

def __init__(self, env):

.....

벨만 기대 방정식을 통해 가치함수를 계산하는 정책 평가

def policy_evaluation(self):

.....

현재 가치 함수에 대해서 탐욕 정책 개선

def policy_improvement(self):

.....

특정 상태에서 정책에 따라 무작위로 행동을 반환

def get_action(self, state):

.....

상태에 따른 정책 반환

def get_policy(self, state):

.....

가치 함수의 값을 반환

def get_value(self, state):

.....

PolicyIteration 클래스:
정책 평가

벨만 기대 방정식을 통해 가치함수를 계산하는 정책 평가

def policy_evaluation(self):

다음 가치함수 초기화

next_value_table = [[0.00] * self.env.width for _ in range(self.env.height)]

모든 상태에 대해서 벨만 기대방정식을 계산

for state in self.env.get_all_states():

value = 0.0

종료 상태의 가치 함수 = 0

if state == [2, 2]:

next_value_table[state[0]][state[1]] = value

continue

벨만 기대 방정식

for action in self.env.possible_actions:

next_state = self.env.state_after_action(state, action)

reward = self.env.get_reward(state, action)

next_value = self.get_value(next_state)

value += (self.get_policy(state)[action] *

(reward + self.discount_factor * next_value))

next_value_table[state[0]][state[1]] = value

self.value_table = next_value_table # 가치함수 테이블 갱신

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s', r} \cancel{p(s', r|s, a)} [r + \gamma v_k(s')]$$

현재 가치 함수에 대해서 탐욕 정책 개선

def policy_improvement(self):

next_policy = self.policy_table

for state in self.env.get_all_states():

if state == [2, 2]: # 종료 상태

continue

value_list = []

반환할 정책 초기화

result = [0.0, 0.0, 0.0, 0.0]

모든 행동에 대해서 [보상 + (할인율 * 다음 상태 가치함수)] 계산

for index, action in enumerate(self.env.possible_actions):

next_state = self.env.state_after_action(state, action)

reward = self.env.get_reward(state, action)

next_value = self.get_value(next_state)

value = reward + self.discount_factor * next_value

value_list.append(value)

받을 보상이 최대인 행동들에 대해 탐욕 정책 개선

max_idx_list = np.argmax(value_list) # argmax(): 조건을 만족하는 인덱스의 배열 반환

가장 큰 가치함수의 행동들의 인덱스 (복수인 경우 모두 포함), 4 x 1 배열

max_idx_list = max_idx_list.flatten().tolist()

4 x 1 배열을 리스트로 변환

prob = 1 / len(max_idx_list)

복수의 행동인 경우 같은 확률로 배분

for idx in max_idx_list:

result[idx] = prob

next_policy[state[0]][state[1]] = result

self.policy_table = next_policy # 정책 테이블 갱신

Policy Iteration 클래스: 정책 개선

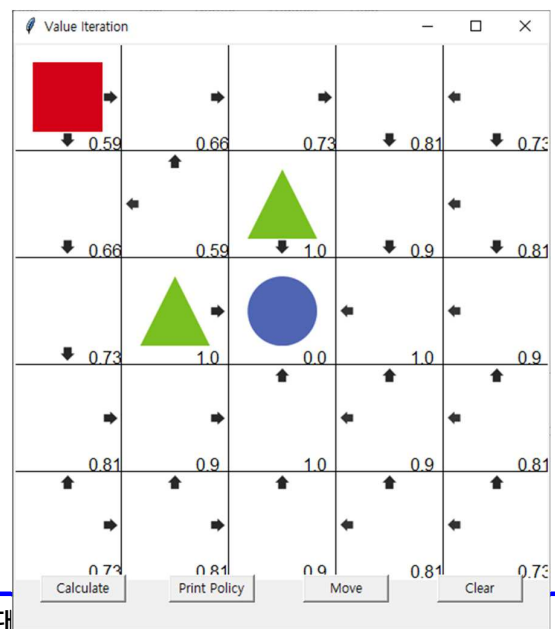
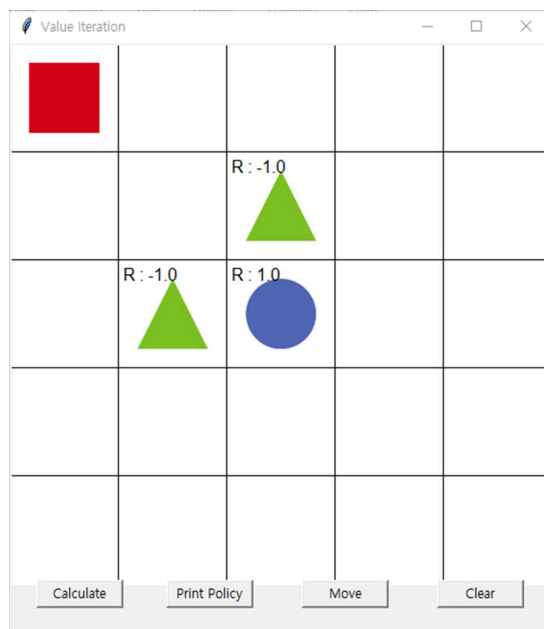
$$\pi'(s) = \underset{a}{\operatorname{argmax}} \sum_{s', r} p(s', r | s, a) [r + \gamma v_{\pi}(s')] \\ = \underset{a}{\operatorname{argmax}} q_{\pi}(s, a)$$

4. 동적 계획법

그리드 월드(Grid World) 예: 가치 반복 (1)

- 5 x 5 그리드 상에서 에이전트(빨간색 사각형)가 목표 지점(파란색 원)로 이동하는 문제

- 버튼: 가치함수 계산, 정책 출력, 이동, 초기화



그리드 월드(Grid World) 예: 가치 반복 (2)

- 5 x 5 그리드 상에서 에이전트(빨간색 사각형)가 목표 지점(파란색 원)로 이동하는 문제
 - 보상
 - 장애물(녹색 삼각형)에 이동 시 -1 보상
 - 목표 지점 도착 시 +1 보상
 - 상태: 그리드 위치 -> [열(x), 행(y)]
 - 가치함수 테이블: **value_table**, (5,5)
 - 행동
 - 각 상태에서의 행동: 좌, 우, 상, 하 -> [0, 1, 2, 3]
 - 정책
 - 각 상태에서의 행동의 확률: 좌/우/상/하 -> [0.25, 0.25, 0.25, 0.25]
 - ~~정책 테이블: **policy_table**, (4,5,5)~~
 - 결정론적 환경
 - 에이전트가 특정 상태에서 특정 행동을 취할 때 전이될 상태의 확률은 1, 나머지 행동의 전이 확률은 0

파일 구성 및 메인: 가치 반복

- 2개의 파일
 - **GridVI.py**
 - 정책 반복 알고리즘 관련 함수(ValueIteration 클래스), 메인
 - **GridVI_environment.py**
 - 그리드월드 GUI 구성(GraphicDisplay 클래스)
 - 상태, 보상 등 환경 정보 제공 함수(Env 클래스)

```
import numpy as np
from environment import GraphicDisplay, Env
.....

if __name__ == "__main__":
    env = Env()          # 그리드 환경 객체 생성
    value_iteration = ValueIteration(env)  # 정책 반복 객체 생성
    # GUI로 그리드 환경 디스플레이
    grid_world = GraphicDisplay(value_iteration)
    grid_world.mainloop()
```

```
# 가치 반복법
class ValueIteration:
    # 생성자: 가치함수 테이블 초기화
    def __init__(self, env):
        .....
    # 벨만 최적 방정식을 통해 가치함수를 계산
    def value_iteration(self):
        .....
    # 현재 가치 함수로부터 행동을 반환
    def get_action(self, state):
        .....
    # 상태에 따른 정책 반환
    def get_policy(self, state):
        .....
    # 가치 함수의 값을 반환
    def get_value(self, state):
        .....
```

Value Iteration 클래스:
가치함수 계산

```
# 벨만 최적 방정식을 통해 가치 함수 계산
def value_iteration(self):
    # 다음 가치함수 초기화
    next_value_table = [[0.0] * self.env.width
                        for _ in range(self.env.height)]

    # 모든 상태에 대해서 벨만 최적방정식을 계산
    for state in self.env.get_all_states():
        # 마침 상태의 가치 함수 = 0
        if state == [2, 2]:
            next_value_table[state[0]][state[1]] = 0.0
            continue

        # 벨만 최적 방정식
        value_list = []
        for action in self.env.possible_actions:
            next_state = self.env.state_after_action(state, action)
            reward = self.env.get_reward(state, action)
            next_value = self.get_value(next_state)
            value_list.append((reward + self.discount_factor * next_value))

        # 최댓값을 다음 가치 함수로 대입
        next_value_table[state[0]][state[1]] = max(value_list)

    self.value_table = next_value_table
```

$$v_{k+1}(s) = \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_k(s')]$$


 $q(s, a)$

현재 가치 함수로부터 행동을 반환

```
def get_action(self, state):
```

```
    if state == [2, 2]:
        return []
```

모든 행동에 대해 큐함수 (보상 + (감가율 * 다음 상태 가치함수))를 계산

```
    value_list = []
```

```
    for action in self.env.possible_actions:
```

```
        next_state = self.env.state_after_action(state, action)
```

```
        reward = self.env.get_reward(state, action)
```

```
        next_value = self.get_value(next_state)
```

```
        value = (reward + self.discount_factor * next_value)
```

```
        value_list.append(value)
```

$$v_{k+1}(s) = \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_k(s')]$$

$q(s, a)$

최대 큐 함수를 가진 행동(복수일 경우 여러 개)을 반환

```
    max_idx_list = np.argwhere(value_list == np.amax(value_list))
```

```
    action_list = max_idx_list.flatten().tolist()
```

```
    return action_list
```

51

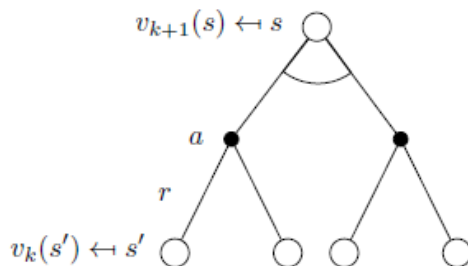
동적 계획법 알고리즘 요약

Problem	Bellman Equation	Algorithm
Prediction	Bellman Expectation Equation	Iterative Policy Evaluation
Control	Bellman Expectation Equation + Greedy Policy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

- Algorithms are based on state-value function $v_{\pi}(s)$ or $v_{*}(s)$
- Complexity $O(mn^2)$ per iteration, for m actions and n states

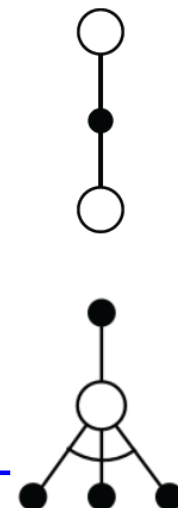
동적 계획법 문제점

- ❑ 마르코프 의사결정 프로세스(MDP)의 **모든 정보**를 알고 있어야 함
- ❑ 각 단계에서의 **가치함수 계산(백업, backup)**
 - 모든 다음 상태와 행동을 고려하여 가치함수 계산 (**full-width backup**)
 - 중간 규모의 문제에는 효과적
 - 수백만개의 상태
 - 많은 상태의 큰 문제의 경우 지수함수적으로 계산량 증가
 - 벨만의 차원 저주 (Bellman's curse of dimensionality)



샘플 백업 (Sample Backup)

- ❑ 이 후에는 **샘플 백업 방식** 소개
 - **강화학습 (Reinforcement learning)**
 - 몬테카를로 학습 (Monte-Carlo Learning)
 - 시차학습 (Temporal Difference Learning)
- ❑ 모든 다음 상태가 아닌 **샘플링된 보상 및 상태 전이** 사용
- ❑ **장점**
 - **비모델 (model-free) 방식**
 - 사전에 MDP를 몰라도 됨
 - 샘플링으로 벨만의 차원의 저주 극복
 - 백업의 비용이 상태 수에 관계 없이 일정



❑ David Silver - UCL Course on RL, 2015

- <https://www.davidsilver.uk/teaching/>
- Lecture 3: Planning by Dynamic Programming

❑ Miguel Morales, Grokking Deep Reinforcement Learning

- <https://livebook.manning.com/book/grokking-deep-reinforcement-learning>
- 그로킹 심층 강화학습, 강찬석 옮김, 한빛미디어
- 3장 목표와 장기 목표 간의 균형
- Chan's Jupyter, 그로킹 심층 강화학습 (Grokking Deep Reinforcement Learning)
 - <https://goodboychan.github.io/book/>

❑ 파이썬과 케라스로 배우는 강화학습, 이응원 외, 위키북스

- <https://github.com/rlcode/reinforcement-learning-kr-v2>