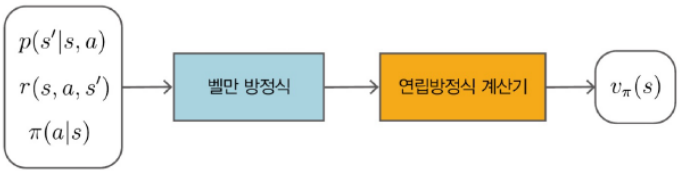
Chapter 4 동적 프로그래밍

벨만 방정식을 이용하면 연립방정식을 얻을 수 있고, 그 연립방정식을 풀 수 있다면 가치 함수를 구할 수 있었다.

그림 4-1 벨만 방정식을 이용하여 가치 함수를 구하는 흐름



이와 같이 상태 전이 확률 $p(s'|s,a)$, 보상 함수 $r(s,a,s')$, 정책 $\pi(a|s)$라는 세 가지 정보가 있다면 벨만 방정식을 이용해 연립방정식을 구할 수 있다. 그리고 연립방정식을 푸는 프로그램을 사용하여 가치 함수를 구할 수 있다. 하지만, 연립방정식은 상태와 행동 패턴의 수가 조금만 많아져도 감당할 수 없게 된다. 그래서 등장한 것이 동적 프로그래밍 (Dynamic Programming) 혹은 동적 계획법이다. 동적 프로그래밍을 이용하면 상태와 행동의 수가 어느 정도 많아져도 가치 함수를 구할 수 있다.

4.1. 동적 프로그래밍과 정책 평가

강화 학습에서는 두 가지 문제를 해결해야 한다. 정책 평가와 정책 제어이다. 정책 평가는 정책 π가 주어졌을 때 그 정책의 가치 함수 $v_\pi(s)$ 또는 $q_\pi(s,a)$를 구하는 문제이다. 정책 제어는 정책을 조정하여 최적 정책을 만들어내는 것을 말한다.

강화 학습의 궁극적인 목표는 정책 제어이다. 대부분 문제에서 최적 정책을 직접 구하기 어렵기 때문에 정책 평가부터 목표로 하는 경우가 많다. 이번 절에서는 동적 프로그래밍 알고리즘을 이용한 정책 평가 방법을 살펴보겠다.

4.1.1. 동적 프로그래밍 기초

앞에서는 가치 함수를 다음과 같이 정의했다.

$$v_\pi(s) = \mathbb{E}_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots | S_t = s]$$

이렇게 무한대가 포함되어 있는 식은 일반적으로 계산할 수 없지만, 벨만 방정식을 이용하여 탈출 할 수 있다.

$$v_\pi(s) = \sum_{a,s'} \pi(a|s)\, p(s'|s,a)\{r(s,a,s') + \gamma v_\pi(s')\}$$

벨만 방정식은 '현재 상태 s의 가치 함수 $v_\pi(s)$'와 '다음 상태 s'이 가치 함수 $v_\pi(s')$의 관계를 나타낸다. DP는 벨만 방정식을 '갱신식'으로 변형한 것에서 파생된다.

$$V_{k+1}(s) = \sum_{a,s'} \pi(a|s)\, p(s'|s,a)\{r(s,a,s') + \gamma V_k(s')\}$$

$V_{k+1}(s)$는 k+1번째로 갱신된 가치 함수를 뜻하며 k번째로 갱신된 가치 함수는 $V_k(s)$로 표기한다. 이때 $V_{k+1}(s)$와 $V_k(s)$는 '추정치'라서 실제 가치 함수인 $v_\pi(s)$와 다르다.

위의 식의 특징은 '다음 상태의 가치 함수 $V_k(s')$를 이용하여 '지금 상태 가치 함수 $V_{k+1}(s)$'를 개신한다는

점이다. 그리고 이 식은 '추정치 $V_k(s')$를 사용하여 '또 다른 추정치 $V_{k+1}(s)$'를 개선한다. 이처럼 추정치를 사용하여 추정치를 계산하는 과정을 부트스트래핑이라고 부른다.

DP를 이용한 구체적인 알고리즘은 다음과 같다. 먼저 $V_0(s)$의 초깃값을 설정한다. 예를 들어 모든 상태에서 $V_0(s) = 0$으로 초기화한다. 그리고 위의 식을 이용하여 $V_0(s)$에서 $V_1(s)$로 갱신한다. 이어서 $V_1(s)$를 기반으로 $V_2(s)$를 갱신한다. 이 일을 반복하다 보면 최종 목표인 $V_\pi(s)$에 가까워진다. 이 알고리즘은 반복적 정책 평가라고 한다.

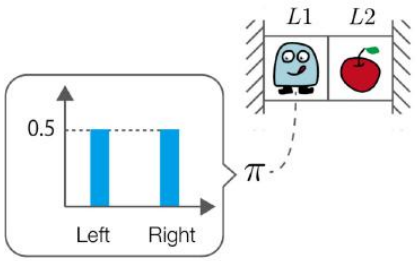
반복적 정책 평가 알고리즘을 실제 문제에 적용하려면 반복되는 갱신을 언젠가는 멈춰야 한다. 이때 갱신 횟수를 결정하는 기준으로 갱신된 양을 이용할 수 있다. 식에 따른 갱신을 반복하면 $V_\pi(s)$에 수렴한다.

cf)이해하기: 반복적 정책 평가
강화학습에서 정책이 주어졌을 때, 그 정책을 따를 경우 각 상태에서 기대되는 보상의 합을 평가하는 과정. 즉, 현재 정책을 따르면 각 상태에서 얼마나 좋은지가 점점 더 정확해지도록 업데이트 하는 과정
초기 가치 함수를 0으로 설정하면서, 점점 갱신하면 현재 정책에서의 진짜 가치 함수인 $V_\pi(s)$에 가까워진다. 벨만 방정식이 수렴성을 보장하고, 각 반복에서 값이 점점 정확해지면서 최종적으로 변화량이 0에 가까워져 수렴하게 된다. (계속 업데이트를 하다 보면 더 이상 값이 변하지 않는 수렴점에 도달하는 것)
-〉 특정한 정책이 주어지면 그 정책을 따를 때 상태의 기대 가치를 정확하게 계산할 수 있게 되는 것
-〉 밸만 방정식은 연립방정식이기 때문에 상태나 행동의 수가 많아질수록 계산이 복잡해 진다. 그래서 동적 프로그래밍 기법(반복적 정책 평가) 기법을 사용하여 점진적으로 업데이트를 하면 최종적으로 정확한 기대 가치를 구할 수 있어 정책을 평가할 수 있게 된다.

4.1.2. 반복적 정책 평가_첫 번째 구현

'두 칸짜리 그리드 월드'를 예로 들어 반복적 정책 평가 알고리즘의 흐름을 이해할 수 있다.

그림 4-2 두 칸짜리 그리드 월드($L1$에서 $L2$로 이동하면 +1 보상, 벽에 부딪히면 −1 보상)



그림에서 에이전트는 무작위 정책 π에 따라 행동한다. (각 행동을 할 확률이 0.5) 이 문제에서 상태 전이는 결정적이다. 수식에서는 다음 상태 s'가 함수 $f(s,a)$에 의해 고유하게 결정된다고 가정한다. 그렇다면 가치 함수 갱신식을 다음과 같이 단순화할 수 있다.

$$V_{k+1}(s) = \sum_{a,s'} \pi(a|s)p(s'|s,a)\{r(s,a,s') + \gamma V_k(s')\} \quad [식\ 4.2]$$

↓ $\sum$를 둘로 구분하여 표기

$$V_{k+1}(s) = \sum_{a} \pi(a|s) \sum_{s'} p(s'|s,a)\{r(s,a,s') + \gamma V_k(s')\}$$

↓ 상태 전이가 결정적

$s' = f(s,a)$ 일 때

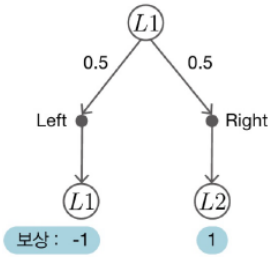
$$V_{k+1}(s) = \sum_{a} \pi(a|s)\{r(s,a,s') + \gamma V_k(s')\} \quad [식\ 4.3]$$

이 식을 이용해서 정책 π의 가치 함수를 갱신할 수 있다. 우선 초깃값으로 $V_0(s)$를 0으로 설정한다. 두 칸짜

리 그리드 월드에서는 상태는 두 개 뿐이므로 다음과 같이 나타낼 수 있다.

$$V_0(L1) = 0$$
$$V_0(L2) = 0$$

이어서 위의 식에 따라 $V_0(s)$를 갱신한다. 백업 다이어그램을 보면 쉽게 이해할 수 있다.



그림과 같이 두 갈래 길로 나뉜다. 하나는 0.5의 확률로 왼쪽으로 가는 행동을 선택해 -1의 보상을 받고 상태는 L1으로 유지된다. 여기에 할인율 γ를 0.9로 가정하고 식에 대입하면 다음과 같다.

$$0.5\{-1+0.9\,V_0(L1)\}$$

그림의 또 다른 가능성은 상태 L1에서 오른쪽으로 가는 행동을 선택한 경우이다. 보상으로 1을 받고 상태 L2로 이동한다.

$$0.5\{1+0.9\,V_0(L2)\}$$

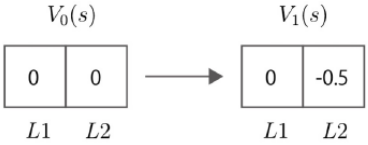
이상으로부터 $V_1(L1)$은 다음과 같이 구할 수 있다.

$$\begin{aligned} V_1(L1) &= 0.5\{-1+0.9\,V_0(L1)\}+0.5\{1+0.9\,V_0(L2)\} \\ &= 0.5(-1+0.9\cdot 0)+0.5(1+0.9\cdot 0) \\ &= 0 \end{aligned}$$

같은 방법으로 $V_1(L2)$도 계산할 수 있다.

$$\begin{aligned} V_1(L2) &= 0.5\{0+0.9\,V_0(L1)\}+0.5\{-1+0.9\,V_0(L2)\} \\ &= 0.5(0+0.9\cdot 0)+0.5(-1+0.9\cdot 0) \\ &= -0.5 \end{aligned}$$

상태는 총 2개 뿐이기 때문에 가치 함수 갱신이 끝났다. 결과를 정리하면 다음과 같다.



그림과 같이 $V_0(s)$를 $V_1(s)$로 갱신했다. 같은 과정을 반복해서 $V_1(s)$에서 $V_2(s)$를 계산하고, $V_2(s)$에서 $V_3(s)$를 계산하는 식으로 반복한다. 파이썬으로 구현할 수 있다.

```python
# 상태 가치 함수 갱신 p.114
V = {'L1':0.0, 'L2': 0.0} # 초깃값을 0으로 설정
new_V = V.copy() # V의 복사본

# 이전 상태 가치 함수를 사용하여 상태 가치 함수 갱신 -> 점점 업데이트하면서 정확한 가치 함수 추정 가능
for _ in range(100):
    new_V['L1'] = 0.5 * (-1 + 0.9 * V['L1']) + 0.5 * (1 + 0.9 * V['L2'])
    new_V['L2'] = 0.5 * (0 + 0.9 * V['L1']) + 0.5 * (-1 + 0.9 * V['L2'])
    V = new_V.copy() # 갱신된 가치 함수를 재저장
print(V)
```

```
{'L1': -2.2499335965027827, 'L2': -2.7499335965027827}
```

실제 가치 함수의 값은 [-2.25, -2.75]이고, 코드를 보면 100번 갱신한 값과 거의 같은 값으로 수렴하는 모습을 볼 수 있다. 임계값을 설정하여 갱신 횟수를 자동으로 결정할 수 있다.

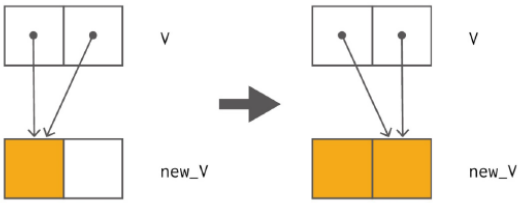
```python
# 임계값을 설정하여 갱신 횟수 자동으로 결정
V = {'L1':0.0, 'L2': 0.0}
new_V = V.copy()

cnt = 0 # 갱신 횟수 기록
while True:
    new_V['L1'] = 0.5 * (-1 + 0.9 * V['L1']) + 0.5 * (1 + 0.9 * V['L2'])
    new_V['L2'] = 0.5 * (0 + 0.9 * V['L1']) + 0.5 * (-1 + 0.9 * V['L2'])

    #갱신된 양의 최댓값
    delta = abs(new_V['L1'] - V['L1'])
    delta = max(delta, abs(new_V['L2'] - V['L2']))

    V = new_V.copy()

    cnt += 1
    if delta < 0.0001: # 임계값 = 0.0001
        print(V)
        print('갱신 횟수:', cnt)
        break
```

```
{'L1': -2.249167525908671, 'L2': -2.749167525908671}
갱신 횟수: 76
```

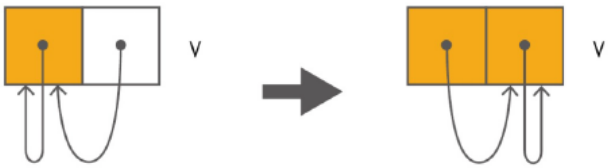
임계값을 0.0001로 설정하고 갱신된 양의 최댓값이 임계값 밑으로 떨어질 때까지 순환문을 반복한다. 출력 결과를 보면 76회 갱신한 끝에 가치 함수의 값이 정답에 근접해져있다.

4.1.3. 반복적 정책 평가_다른 구현 방법

위에서는 반복적 정책 평가 알고리즘을 구현하기 위해 두 개의 딕셔너리를 사용했다. 하나는 현재의 가치 함수를 보관하는 V, 다른 하나는 갱신 시 사용하는 new_V이다. 이 두 함수를 사용하여 가치 함수를 갱신했다.



new_V의 각 원소를 계산할 때 V라는 딕셔너리의 값을 사용하고 있다는 점이다. 다르게 구현할 수도 있다. 바로 딕셔너리를 하나만 쓰고 각 원소를 '덮어쓰는' 방식이다.



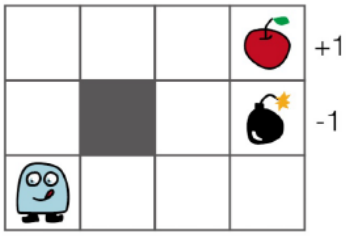
이 방식을 사용하면 갱신한 원소를 곧바로 재활용하기 때문에 더 빠르다. (왼쪽의 이미 갱신된 값을 바로 사용하여 오른쪽 원소를 갱신) 코드로 구현하면 다음과 같다.

```
# 덮어쓰기 방식으로 구현
V = {'L1':0.0, 'L2': 0.0}

cnt = 0
while True:
    t = 0.5 * (-1 + 0.9 * V['L1']) + 0.5 * (1 + 0.9 * V['L2'])
    delta = abs(t - V['L1'])
    V['L1'] = t # 바로 덮어쓰기

    t = 0.5 * (0 + 0.9 * V['L1']) + 0.5 * (-1 + 0.9 * V['L2'])
    delta = max(delta, abs(t - V['L2']))
    V['L2'] = t

    cnt += 1
    if delta < 0.001:
        print(V)
        print('갱신 횟수:', cnt)
        break
```

```
{'L1': -2.2441903310332854, 'L2': -2.7445822014263284}
갱신 횟수: 44
```

4.2. 더 큰 문제를 향해

반복적 정책 평가 알고리즘을 이용하면 상태와 행동 패턴의 수가 어느 정도 많아져도 빠르게 풀 수 있다. 이번에는 '3*4 그리드 월드'의 문제를 살펴 보자.



설정은 다음과 같다.

- 에이전트는 상하좌우 네 방향으로 이동할 수 있다.
- [그림 4-8]에서 회색 칸은 벽을 뜻하며 벽 안으로는 들어갈 수 없다.
- 그리드 바깥도 벽으로 둘러싸여 더 이상 나아갈 수 없다.
- 벽에 부딪히면 보상은 0이다.
- 사과는 보상 +1, 폭탄은 보상 -1, 그 외의 보상은 0이다.
- 환경의 상태 전이는 고유하다(결정적). 즉, 에이전트가 오른쪽으로 이동하는 행동을 선택하면 (벽만 없다면) 반드시 오른쪽으로 이동한다.
- 이번 문제는 일회성 과제로서, 사과를 얻으면 종료한다.

4.2.1. GridWorld 클래스 구현

```python
# GridWorld 클래스 구현
import numpy as np

class GridWorld:
    def __init__(self):
        self.action_score = [0, 1, 2, 3] # 행동 공간(가능한 행동들)
        self.action_meaning = { # 행동의 의미
            0: 'UP',
            1: 'DOWN',
            2: 'LEFT',
            3: 'RIGHT',
        }
        self.reward_map = np.array( # 보상 맵 (각 좌표의 보상 값)
            [[0,0,0,1.0],
             [0,None,0,-1.0],
             [0,0,0,0]]
        )
        self.goal_state = (0,3) # 목표 상태(사과 좌표)
        self.wall_state = (1,1) # 벽 좌표
        self.start_state = (2,0) # 시작 상태
        self.agent_state = self.start_state # 에이전트 초기 상태
```

self.reward_map을 활용하여 각 좌표로 이동했을 때 얻는 보상의 크기를 담아냈다. 또한, 이번 문제는 일회성 과제이기 때문에 에이전트가 목표 상태에 도달하면 문제가 끝이 난다.

이어서 GridWorld 클래스의 메서드들을 살펴볼 수 있다.

```python
    # GridWorld 클래스의 메서드들

    @property
    def height(self):
        return len(self.reward_map)

    @property
    def width(self):
        return len(self.reward_map[0])

    @property
    def shape(self):
        return self.reward_map.shape

    def actions(self):
        return self.action_space

    def states(self):
        for h in range(self.height):
            for w in range(self.width):
                yield (h,w)
```

```python
env = GridWorld()

print(env.height)
print(env.width)
print(env.shape)
```

```
3
4
(3, 4)
```

actions()와 states()의 메서드들을 사용하면 모든 행동과 모든 상태에 순차적으로 접근할 수 있다.

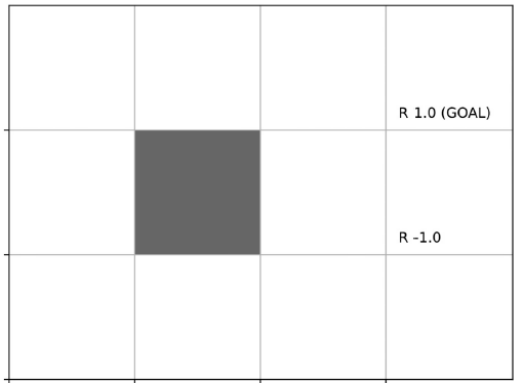
```python
for action in env.actions(): # 모든 행동에 순차적으로 접근
    print(action)
print("----------------------")
for state in env.states(): # 모든 상태에 순차적으로 접근
    print(state)
```

```
0
1
2
3
----------------------
(0, 0)
(0, 1)
(0, 2)
(0, 3)
(1, 0)
(1, 1)
(1, 2)
(1, 3)
(2, 0)
(2, 1)
(2, 2)
(2, 3)
```

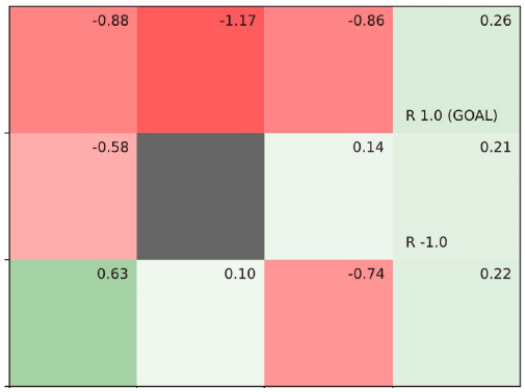
이어서 환경의 상태 전이를 나타내는 메서드인 next_state()와 보상 함수 메서드인 reward()를 구현할 수 있다.

```python
# 상태 전이 next_state()와 보상 함수 reward()
def next_state(self, state, action):
    # 이동 위치 계산
    action_move_map = [(-1, 0), (1, 0), (0, -1), (0, -1)] # 상하좌우
    move = action_move_map[action]
    next_state = (state[0] + move[0], state[1] + move[1])
    ny, nx = next_state

    # 이동한 위치가 그리드 월드의 테두리 밖이나 벽이면 이동 하지 않는다.
    if nx < 0 or nx >= self.width or ny < 0 or ny > self.height:
        next_state = state
    elif next_state == self.wall_state:
        next_state = state

    return next_state

def rewarod(self, state, action, next_state):
    return self.reward_map[next_state] # 다음 상태 만으로 보상값을 결정
```

또한 GridWorld 클래스에는 그리드 월드를 시각화 할 수 있다. (env.render_v())



벽은 회색으로 칠해져 있고, 각 칸의 왼쪽 아래에는 보상값이 적혀져 있다. 이 render_v() 메서드는 상태 가치 함수를 매개변수로 받을 수 있다.



가치 함수를 건네면 각 위치의 가치 함수 값이 해당 칸 오른쪽 위에 표시된다. 그리고 값의 크기에 따라 색을 달리해서 작을수록 붉은색이 짙어지고 클수록 녹색이 짙어진다.

한편 행동 가치 함수를 위한 시각화 함수인 render_q()메서드도 제공한다.

GridWorld 클래스에는 이 외에도 reset()과 step(action) 메서드가 있다. reset()은 게임을 초기 상태로 되돌리는 메서드이다. 에이전트를 시작 위치로 되돌려 준다. step(action)은 에이전트에게 action이라는 행동을 시켜 한 단계 진행시킨다. (하지만, 반복적 정책 평가 알고리즘에서는 에이전트에게 행동을 시키지 않기

때문에 이 두 메서드는 사용하지 않는다.)

4.2.2. defaultdict 사용법

지금까지 가치 함수를 딕셔너리로 구했다.

```
from common.gridworld import GridWorld

env = GridWorld()
V = {}

# 딕셔너리 원소 초기화
for state in env.states():
    V[state] = 0

state = (1, 2)
print(V[state])  # 상태 (1,2)의 가치 함수 출력
```

딕셔너리는 V[key]형태로 사용하는데, 딕셔너리 안에 key가 존재하지 않으면 오류가 난다. 따라서 앞의 코드처럼 모든 원소를 초기화해야 한다. 이러한 번거러움을 덜어주기 위해 파이썬 라이브러리 defaultdict를 사용할 수 있다.

```
from collections import defaultdict  # defaultdict 임포트
from common.gridworld import GridWorld

env = GridWorld()
V = defaultdict(lambda: 0)

state = (1, 2)
print(V[state])  # [출력 결과] 0
```

만약 딕셔너리에 존재하지 않는 키를 건네면 (주어진 키, 기본값) 형태의 새 원소를 새로 만들어 넣는다.

무작위 정책을 defaultdict를 사용하여 구현할 수 있다. '3*4 그리드 월드' 문제에서 에이전트가 취할 수 잇는 행동은 네 개이며, 각 행동은 [0,1,2,3]으로 표현된다. 이 네 개의 행동이 균일하게 무작위로 선택된다면 각 행동이 수행될 확률은 모두 0.25이다. 따라서 행동의 확률 분포는 {0:0.25, 1:0.25, 2:0.25, 3:0.25}로 나타낼 수 있으며, 무작위 정책은 다음처럼 구현할 수 있다.

```
pi = defaultdict(lambda: {0: 0.25, 1: 0.25, 2: 0.25, 3: 0.25})

state = (0, 1)
print(pi[state])  # [출력 결과] {0:0.25, 1:0.25, 2:0.25, 3:0.25}
```

Pi는 상태가 주어지면 해당 상태에서 취할 수 잇는 행동의 확률 분포를 반환한다. 출력 결과는 모든 행동이 똑같이 0.25 확률로 선택되는 분포이다.

4.2.3. 반복적 정책 평가 구현

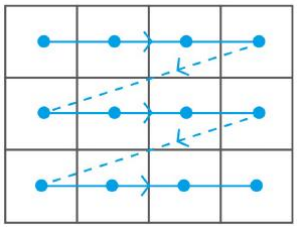
반복적 정책 평가 알고리즘을 구현해볼 수 있다. 우선 갱신을 한 단계만 수행하는 함수를 구현할 수 있다. 여기서 구현할 eval_onestep() 함수는 다음과 같이 네 개의 매개변수를 받는다.

- pi(defaultdict): 정책
- V(defaultdict): 가치 함수
- env(GridWorld): 환경
- gamma(float): 할인율

ch04/policy_eval.py

```
def eval_onestep(pi, V, env, gamma=0.9):
    for state in env.states():  # ❶ 각 상태에 접근
        if state == env.goal_state:  # ❷ 목표 상태에서의 가치 함수는 항상 0
            V[state] = 0
            continue

        action_probs = pi[state]  # probs는 probabilities(확률)의 약자
        new_V = 0

        # ❸ 각 행동에 접근
        for action, action_prob in action_probs.items():
            next_state = env.next_state(state, action)
            r = env.reward(state, action, next_state)
            # ❹ 새로운 가치 함수
            new_V += action_prob * (r + gamma * V[next_state])

        V[state] = new_V
    return V
```

eval_onestep() 함수는 for문을 이중으로 사용하고 있다. 바깥 for문인 1에서는 모든 상태를 하나씩 추적한다.



그림처럼 모든 상태에 순차적으로 접근한다. 그리고 2에 상태 state가 목표 상태이면 가치 함수를 0으로 설정한다. 에이전트가 목표 지점에 도달하면 에피소드가 끝나고 그 다음 전개는 없기 때문이다. 따라서 목표 상태에서의 가치 함수 값은 항상 0이다.

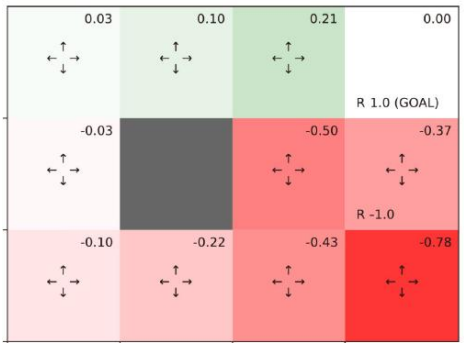
3에서는 행동의 확률 분포를 가져온다. 그리고 상태 전이 함수로 다음 상태를 얻는다. 이 eval_onestep() 함수로 가치 함수가 한 차례 갱신된다. 이제부터는 이 갱신을 반복해야 하는데, 코드로 구현할 수 있다.

```
def policy_eval(pi, V, env, gamma, threshold=0.001):
    while True:
        old_V = V.copy()  # 갱신 전 가치 함수
        V = eval_onestep(pi, V, env, gamma)

        # 갱신된 양의 최댓값 계산
        delta = 0
        for state in V.keys():
            t = abs(V[state] - old_V[state])
            if delta < t:
                delta = t

        # 임곗값과 비교
        if delta < threshold:
            break
    return V
```

eval_onestep() 함수를 반복 호출하여 갱신된 양의 최댓값이 임계값보다 작아지면 갱신을 중단한다. 이를 활용하여 정책 평가를 수행할 수 있다.

```
pi = defaultdict(lambda: {0: 0.25, 1: 0.25, 2: 0.25, 3: 0.25})  # 정책
V = defaultdict(lambda: 0)  # 가치 함수

V = policy_eval(pi, V, env, gamma)  # 정책 평가
env.render_v(V, pi)  # 시각화
```

이 코드를 실행하면 다음 그림을 얻을 수 있다.



무작위 정책의 가치 함수를 보여준다. 예를 들어 시작점의 왼쪽 맨 아래칸의 가치 함수는 -0.10이다. 시작점에서 무작위로 움직이면 기대 수익이 -0.10이라는 뜻이다. 에이전트가 무작위로 움직이기 때문에 폭탄을 얻을 수도 있다. 값이 - 값이라 사과보다 폭탄을 얻을 확률이 조금 더 크다. 또한 맨 아래 줄과 가운데 줄은 모두 마이너스인데, 이 위치들에서는 폭탄의 영향이 더 크다는 뜻이다.

4.3. 정책 반복법

강화학습의 최종 목표는 최적 정책 찾기이다. 벨만 방정식을 사용하는 방법이 있지만, 계산량이 너무 많다는 단점이 잇다. 그래서 DP를 사용하여 정책을 평가할 수 있다. 평가를 통해 정책을 수정하고 더 나아지는지 비교 하며 개선할 수도 있다.

4.3.1. 정책 개선

정책을 개선하는 힌트는 '최적 정책'에서 찾을 수 있다. 이번 절에서는 다음 기호를 사용하여 정책 개선 방법을 설명한다.

- 최적 정책: $\mu_*(s)$
- 최적 정책의 상태 가치 함수: $v_*(s)$
- 최적 정책의 행동 가치 함수(Q 함수): $q_*(s,a)$

최적 정책 μ_*은 다음 식으로 표현된다.

$$\mu_*(s) = \underset{a}{\operatorname{argmax}}\, q_*(s,a)$$
$$= \underset{a}{\operatorname{argmax}} \sum_{s'} p(s' \mid s,a)\{r(s,a,s') + \gamma v_*(s')\}$$

위 식은 최적 정책에 대한 식이지만, 여기서는 '임의의 결정적 정책' μ에 식을 적용할 수 있다.

$$\mu'(s) = \underset{a}{\operatorname{argmax}}\, q_\mu(s,a)$$
$$= \underset{a}{\operatorname{argmax}} \sum_{s'} p(s' \mid s,a)\{r(s,a,s') + \gamma v_\mu(s')\}$$

이때 각 개념을 다음 기호로 표기합니다.

- 현 상태의 정책: $\mu(s)$
- 정책 $\mu(s)$의 상태 가치 함수: $v_\mu(s)$
- 새로운 정책: $\mu'(s)$

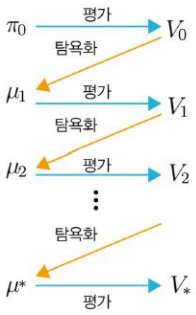
모든 상태 s에서 $\mu(s)$와 $\mu'(s)$가 같다면 (정책이 그대로라면), 정책 $\mu(s)$은 이미 최적 정책이라는 것이다. 왜냐하면 식에 의해 정책이 갱신되지 않는다면 다음 식이 만족하기 때문이다.

$$\mu(s) = \underset{a}{\operatorname{argmax}}\, q_{\mu}(s,a)$$

따라서 모든 상태 s에서 $\mu'(s)$가 갱신되지 않는다면 $\mu(s)$는 이미 최적 정책이라는 뜻이다. 탐욕화의 결과로 정책이 갱신되는 경우에서는 정책 μ'가 정책 μ와 달라진다면 새로운 정책은 항상 기존 정책보다 개선된다. 즉, 모든 상태 s에서 $v_{\mu'}(s) \geq v_{\mu}(s)$가 성립한다.

4.3.2. 평가와 개선 반복

앞 절에서 탐욕화로 정책을 개선할 수 있었다. 그렇다면 평가와 탐욕화가 최적 정책을 찾는 방법의 핵심이다.



글로 정리하면 다음과 같다.

1. 먼저 π_0이라는 정책에서 시작한다. 정책 π_0은 확률적일 수도 있으므로 $\mu_0(s)$가 아닌 $\pi_0(s|a)$로 표기한다.
2. 다음으로 정책 π_0의 가치 함수를 평가하여 V_0을 얻는다. 반복적 정책 평가 알고리즘을 이용하면 된다.
3. 그리고 가치 함수 V_0을 이용하여 탐욕화를 수행한다([식 4.7]을 적용하여 정책 갱신). 탐욕 정책은 언제나 하나의 행동을 선택하므로 결정적 정책인 μ_1을 얻을 수 있다.
4. 1~3 과정을 반복한다.

이 과정을 계속하면 탐욕화를 해도 정책이 더 이상 갱신되지 않는 지점에 도달한다. 그때의 정책이 최적 정책, 최적 가치 함수이다. 이렇게 평가와 개선을 반복하는 알고리즘을 정책 반복법이라고 한다.

4.4. 정책 반복법 구현

정책 반복법을 이용하여 최적 정책을 찾을 수 있다. '3*4 그리드 월드'를 문제로 가정해볼 수 있다. 앞에서는 정책을 평가했기 때문에 '정책 개선'을 하면 된다.

4.4.1. 정책 개선

정책을 개선하기 위해서는 현재의 가치 함수에 대한 탐욕 정책을 구한다.

$$\mu'(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} p(s' \mid s,a)\{r(s,a,s') + \gamma v_{\mu}(s')\}$$

또한, 이번 문제에서는 상태 전이가 결정적이기 때문에 탐욕화를 단순화 할 수 있다.

$$s' = f(s,a) \text{ 일 때}$$

$$\mu'(s) = \underset{a}{\operatorname{argmax}}\{r(s,a,s') + \gamma v_{\mu}(s')\}$$

식과 같이 다음 상태 s'는 하나만 존재할 수 있다. Argmax() 함수를 사용하여 가치 함수를 탐욕화하는 함수를 구할 수 있다.

```
def greedy_policy(V, env, gamma):
    pi = {}

    for state in env.states():
        action_values = {}

        for action in env.actions():
            next_state = env.next_state(state, action)
            r = env.reward(state, action, next_state)
            value = r + gamma * V[next_state]  # ❶
            action_values[action] = value

        max_action = argmax(action_values)  # ❷
        action_probs = {0: 0, 1: 0, 2: 0, 3: 0}
        action_probs[max_action] = 1.0
        pi[state] = action_probs  # ❸
    return pi
```

1에서는 각 행동을 대상으로 위의 식의 $r(s,a,s') + rv_{\pi}(s')$부분을 계산한다. 그리고 2에서 argmax() 함수를 호출하여 가치 함수 값이 가장 큰 행동을 찾은 다음, max_action이 선택될 확률이 1.0이 되도록(결정적이 되도록) 확률 분포를 생성한다. 그리고 이를 상태 state에서 취할 수 있는 행동의 확률 분포로 설정한다.

4.4.2. 평가와 개선 반복

정책 반복법을 policy_iter()함수로 구현할 수 있다.

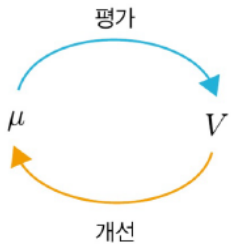
```
ch04/
def policy_iter(env, gamma, threshold=0.001, is_render=False):
    pi = defaultdict(lambda: {0: 0.25, 1: 0.25, 2: 0.25, 3: 0.25})
    V = defaultdict(lambda: 0)

    while True:
        V = policy_eval(pi, V, env, gamma, threshold)  # ❶ 평가
        new_pi = greedy_policy(V, env, gamma)          # ❷ 개선

        if is_render:
            env.render_v(V, pi)

        if new_pi == pi:  # ❸ 갱신 여부 확인
            break
        pi = new_pi

    return pi
```

먼저 정책 pi와 가치 함수 v를 초기화한다.각 정책 pi의 초깃값은 각 행동이 균등하게 선택되도록 설정한다.

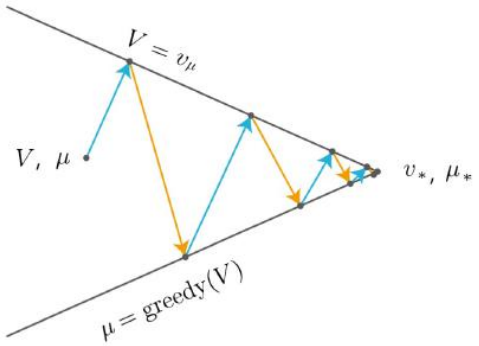
1에서는 현재 정책을 평가하여 가치 함수 V를 얻는다. 그다음 2에서 V를 바탕으로 탐욕화된 정책 new_pi를 얻는다. 3에서는 정책이 갱신되었는지 확인한다. 갱신되지 않는다면 벨만 최적 방정식을 만족하는 것이고, 이때의 pi가 최적 정책이라는 뜻이다.

4.5. 가치 반복법

정책 반복법은 '평가'와 '개선'이라는 두 과정을 번갈아 반복하는 것이다.



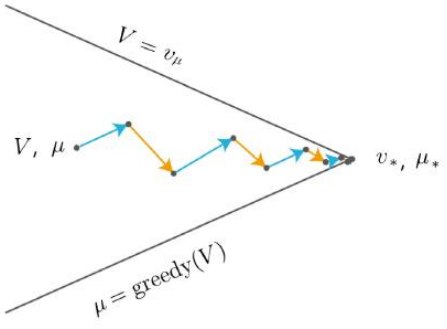
'평가'단계에서는 정책을 평가하여 가치 함수를 얻는다. 그리고 '개선' 단계에서는 가치 함수를 탐욕화하여 개선된 정책을 얻는다. 이를 번갈아 반복하면서 최적 정채과 최적 가치 함수에 점점 가까워진다. 그림으로 표현하면 다음과 같다.



위쪽 직선은 $V = v_\pi$를 나타내며, 임의의 가치 함수와 정책의 살제 가치 함수가 일치하는 곳이다. 아래쪽 직선은 가치 함수를 탐욕화하여 얻은 정책과 μ가 일치하는 곳이다.

정책 반복법은 '평가'와 '개선'을 번갈아 반복한다. '평가' 단계에서는 정책 μ를 평가하여 V_μ를 얻는데, 그림에서 직선 위로 이동하는것에 해당한다. 반면 '개선' 단게에서는 V를 탐욕화 한다. 그림에서 밑의 직선 위를 이동하는 것에 해당한다. 이 두작업을 번갈아 반복하면서 V와 μ가 갱신되고 최적에 도달한다.

정책 반복법은 위의 그림처럼 목표에 도달하기 위해 두 직선 사이를 지그재그로 이동한다.

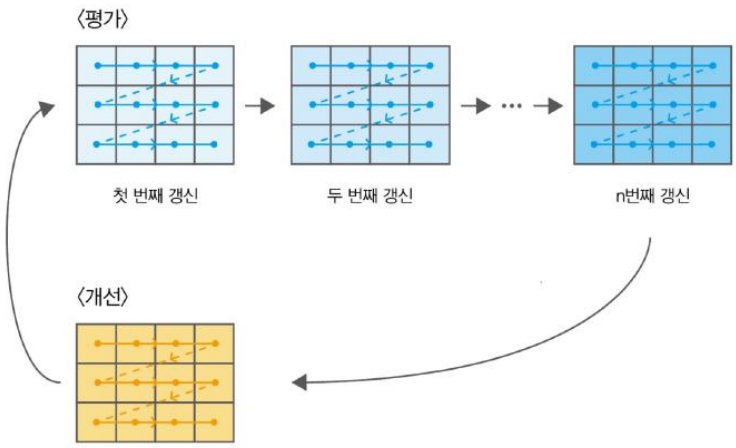


다음 그림과 같은 궤도도 생각해볼 수 있다. 지그재그 궤적을 그리고 있지만, 두 직선에 닿기 전에 방향을 전환한다. '평가'를 완전히 끝내기 전에 '개선' 단계로 전환하고, '개선'을 완전히 끝내기 전에 '평가' 단계로 전환하면서 그림과 같은 궤적을 만들어낸다. 이를 일반화한 정책 반복이라고 한다.

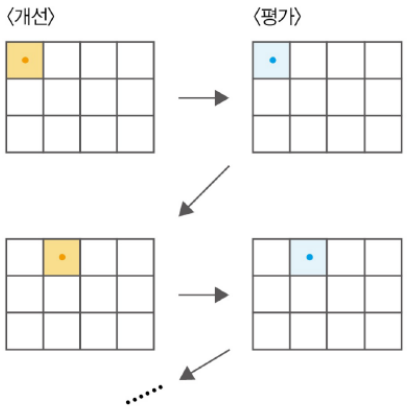
정책 반복법에서는 '평가'와 '개선'을 완벽하게 수행했다. 평가와 개선을 '최대한'으로 번갈아 수행한다. 하지만 평가와 개선을 각각 '최소한'으로 수행하면 어떨까? 하는게 가치 반복법의 아이디어다.

4.5.1. 가치 반복법 도출

정책 반복법의 평가 단계에서는 그림처럼 반복적으로 가치 함수를 갱신한다.



그림과 같이 모든 상태의 가치 함수를 여러 번 갱신한다. 이 갱신 작업이 수렴되면 '개선(탐욕화)' 단계로 넘어간다. 가치 반복법은 하나의 상태만 딱 한번 갱신하고 곧바로 개선 단계로 넘어갈 수 있다.



그림처럼 상태 하나만 개선하고 곧장 평가 단계로 넘어간다. 평가 단계에서도 해당 상태 하나의 가치 함수를 한 번만 갱신한다. 그런 다음 다은 위치를 개선하고 평가하는 흐름으로 진행된다.

수식으로 정리할 수 있다. 개선 단계에서 탐욕화하는 수식은 다음과 같이 쓸 수 있다.

$$\mu(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} p(s' \mid s,a)\{r(s,a,s') + \gamma V(s')\}$$

현재 상태와 보상, 그리고 다음 상태를 활용하여 argmax로 계산한다. argmax는 하나의 행동을 선택해주므로 μ(s)와 같이 결정적 정책으로 표현할 수 있다.

다음은 평가 단계이다. 갱신 전의 가치 함수를 V(s), 갱신 후의 가치 함수를 V'(s)라고 하면 DP에 의한 갱신식은 다음과 같이 표현된다.

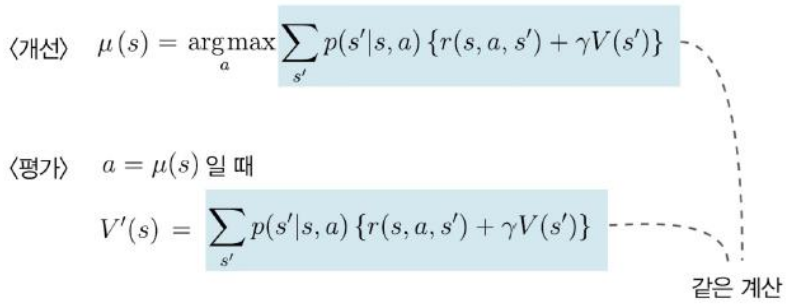
$$V'(s) = \sum_{a,s'} \pi(a \mid s)\, p(s' \mid s,a)\{r(s,a,s') + \gamma V(s')\}$$

이 식에서는 정책이 확률적 정책으로 표기되어 있다. 그러나 '개선' 단계를 한 번 거치면 정책이 탐욕 정책으로 바뀐다. 따라서 결정적 정책으로 바꿔 단순화 할 수 있다.

$a = \mu(s)$ 일 때

$$V'(s) = \sum_{s'} p(s' \mid s,a)\{r(s,a,s') + \gamma V(s')\}$$

이것이 평가 단계의 함수 갱신식이다. 개선과 평가 단계의 식을 보면 다음과 같다.



그림을 보면 개선과 평가 단계에서 똑같이 계산이 중복됨을 알 수 있다. 개선 단계에서 탐욕 행동을 찾기 위해 계산을 한 후, 평가 단계에서 그 탐욕 행동을 이용하여 똑 같은 계산을 다시 수행한다. 이 중복된 계산은 하나로 묶을 수 있다.

$$V'(s) = \max_a \sum_{s'} p(s' \mid s,a)\{r(s,a,s') + \gamma V(s')\}$$

식에서는 최댓값을 찾아주는 max 연산자를 사용하여 가치 함수를 직접 갱신한다.

위 식에서는 정책 μ가 등장하지 않는다. 즉, 정책을 사용하지 않고 가치 함수를 갱신하고 있다. 이를 활용하여 가치 함수를 구하는 알고리즘을 '가치 반복법'이라고 부른다. 가치 반복법으 s이 하나의 식만을 이용하여 평가와 개선을 동시에 수행한다.

또한, 갱신식을 다음 형태로도 표현할 수 있다.

$$V_{k+1}(s) = \max_a \sum_{s'} p(s' \mid s,a)\{r(s,a,s') + \gamma V_k(s')\}$$

가치 반복법은 k= 0부터 시작하여 순서대로 가치 함수를 갱신하는 알고리즘으로, DP의 특징인 같은 계산을 두 번 하지 않는다는 조건을 만족한다.

가치 반복법으로 갱신을 무한히 반복하면 최적 가치 함수를 얻을 수 있다. 멈추기 위해서는 임계값을 정해놓고, 모든 상태의 갱신량이 임계값 밑으로 떨어지면 갱신을 중단한다. 최적의 가치함수가 정해지면 최적 정채책은 다음 식으로 구할 수 있다.

$$\mu_*(s) = \arg\max_a \sum_{s'} p(s' \mid s,a)\{r(s,a,s') + \gamma V_*(s')\}$$

4.5.2. 가치 반복법 구현

'3*4 그리드 월드'문제를 통해 가치 반복법을 구현할 수 있다. 이 문제에서는 상태 전이가 결정적이기 때문에 가치 함수의 갱신식을 단순화할 수 있다.

$$V'(s) = \max_a \sum_{s'} p(s'|s,a)\{r(s,a,s') + \gamma V(s')\} \quad [식\ 4.11]$$

↓ 상태 전이가 결정적

$s' = f(s,a)$ 일 때

$$V'(s) = \max_a \{r(s,a,s') + \gamma V(s')\} \quad [식\ 4.13]$$

식을 따라 한번만 갱신하는 value_iter_onestep() 함수를 구현할 수 있다.

```
ch04
def value_iter_onestep(V, env, gamma):
    for state in env.states():       # ❶ 모든 상태에 차례로 접근
        if state == env.goal_state:  # 목표 상태에서의 가치 함수는 항상 0
            V[state] = 0
            continue

        action_values = []
        for action in env.actions():  # ❷ 모든 행동에 차례로 접근

            next_state = env.next_state(state, action)
            r = env.reward(state, action, next_state)
            value = r + gamma * V[next_state]  # ❸ 새로운 가치 함수
            action_values.append(value)

        V[state] = max(action_values)  # ❹ 최댓값 추출
    return V
```

1에서 모든 상태에 순서대로 접근하고 2에서는 모든 행동에 순서대로 접근한다. 3에서는 식의 중괄호 안쪽을 계산하고 4에서는 max() 함수로 최댓값을 찾아 V[state]를 갱신한다.

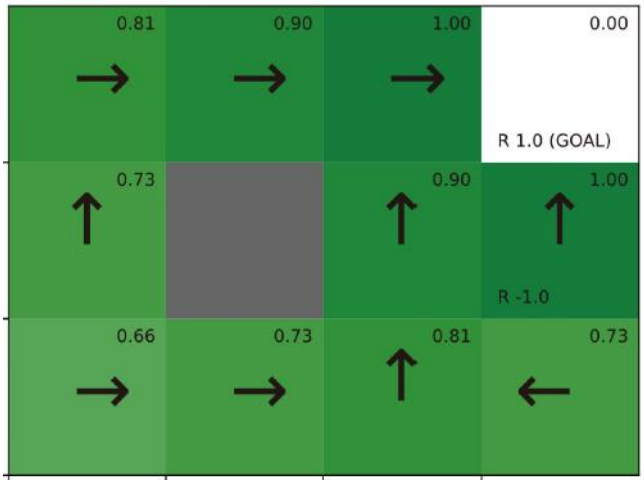
이 작업을 수렴할 때까지 반복 호출하면 된다.

```
def value_iter(V, env, gamma, threshold=0.001, is_render=True):
    while True:
        if is_render:
            env.render_v(V)

        old_V = V.copy()  # 갱신 전 가치 함수
        V = value_iter_onestep(V, env, gamma)

        # 갱신된 양의 최댓값 구하기
        delta = 0
        for state in V.keys():
            t = abs(V[state] - old_V[state])
            if delta < t:
                delta = t

        # 임곗값과 비교
        if delta < threshold:
            break
    return V
```

가치 함수의 갱신량 최댓값이 임계값보다 작아질 때까지 계속 갱신한다. 또한 value_iter()로 갱신되는 가치 함수의 값을 그래프로 그릴 수 있다.

```
from common.gridworld import GridWorld
from ch04.policy_iter import greedy_policy

V = defaultdict(lambda: 0)
env = GridWorld()
gamma = 0.9

V = value_iter(V, env, gamma)  # 최적 가치 함수 찾기

pi = greedy_policy(V, env, gamma)  # 최적 정책 찾기
env.render_v(V, pi)
```

Value_iter() 함수로 최적 가치 함수를 구한다. 최적 가치 함수를 알면 이를 탐욕화하여 최적 정책을 얻을 수 있다. 이를 통해 그림과 같이 최적 정책을 얻을 수 있다.



이 그림이 최적 상태 가치 함수를 탐욕화하여 얻은 정책이다.

cf)이해하기: 정책 반복법과 가치 반복법

1. 정책 반복법
: 정책을 반복적으로 개선하면서 최적 정책을 찾아가는 방법 (현재 정책을 평가하고 더 나은 정책으로 개선하는 과정을 반복)

- 정책 평가: 현재 정책 π가 주어졌을 때, 정확한 상태 가치 함수 $V_\pi(s)$를 계산, 반복적 정책 평가를 사용해서 $V_\pi(s)$가 수렴할 때까지 업데이트
- 정책 개선: 상태 s에서 기존 정책이 선택하는 행동 π(s)보다 더 나은 행동이 있는 지 확인하고, 있다면 정책을 업데이트
- 정책 반복: 정책이 더 이상 바뀌지 않을 때까지 정책 평가 -〉 개선을 반복

2. 가치 반복법
: 정책 반복법보다 더 효율적인 방법, 정책을 직접 평가하는 대신 가치 함수 $V_\pi(s)$만 반복적으로 업데이트해서 최적 정책을 찾는 방법

- 가치 함수 갱신: 벨만 최적 방정식을 사용해서 가치 함수 업데이트. 정책을 따르면서 행동을 평가하는 것이 아니라, 각 상태에서 가능한 모든 행동을 고려하여 최적 값을 찾음. 정책 평가 단계를 건너뛰고, 가치 함수를 바로 업데이트 하므로 더 빠르게 수렴할 수 있다.
- 정책 추출: 최적 가치 함수가 구해지면, 그걸 이용해서 최적 정책을 구한다. 즉, 각 상태에서 가치가 가장 높은 행동을 선택하는 정책을 만든다.

🔥 정책 반복법 vs 가치 반복법 비교

	정책 반복법 (Policy Iteration)	가치 반복법 (Value Iteration)
💡 핵심 아이디어	정책을 평가하고, 개선하는 과정 반복	가치 함수만 업데이트하면서 최적 정책 찾음
🔁 과정	**① 정책 평가 → ② 정책 개선 → ③ 반복**	**① 가치 업데이트 → ② 정책 추출**
⛹️ 속도	비교적 느림 (정책 평가를 여러 번 해야 함)	더 빠름 (정책 평가 없이 가치만 반복 업데이트)
📌 언제 사용?	작은 상태 공간에서 최적 정책을 명확히 찾고 싶을 때	큰 상태 공간에서 빠르게 최적 정책을 찾고 싶을 때