강화학습

동적프로그래밍

고려대학교 세종캠퍼스 인공지능사이버보안학과
구 자 훈

목차

PREVIEW

- 이웃과 연결성
  - 사람은 이웃으로부터 영향을 받고 이웃에 영향을 끼치며 살아감
  - 강화 학습에서는 상태와 상태, 행동과 행동, 정책과 정책 사이에 밀접한 연결성
  - 강화 학습 알고리즘은 연결성을 활용하여 이웃과 정보를 주고 받으며 효율적으로 학습

그림 3-1 축구 응원에서 이웃과 연결성

- 동적 프로그래밍은 초창기에 주로 사용된 오래된 알고리즘
  - 현재는 잘 쓰이지 않지만, 현대 학습 알고리즘의 개념과 기초 공식을 제공함

3.1 원리

- 동적 프로그래밍dynamic programming은 알고리즘 방법론
  - 알고리즘 방법론에는 동적 프로그래밍, 분할 정복, 탐욕 알고리즘, 백트래킹, 한정 분기 등
- 동적 프로그래밍은 상향식 방법
  - 크기가 n인 문제를 크기가 1인 가장 작은 문제로 분해
  - 크기가 1인 문제의 답을 표에 기록
  - 크기가 1인 문제의 답을 보고 크기 2인 문제의 답을 구해 표에 기록
  - 크기가 2인 문제의 답을 보고 크기 3인 문제의 답을 구해 표에 기록
  - 이런 과정을 반복하다가 원래 문제의 크기 n의 답을 얻으면 멈춤

3.1 원리

- [예제 3-1] 동적 프로그래밍을 이용한 최적 행렬 곱셈 순서 정하기
  - n개 행렬 곱셈 $\mathbf{A}_1 \times \mathbf{A}_2 \times \cdots \times \mathbf{A}_n$을 어떤 순서로 곱해야 가장 효율적인지 알아내는 문제
    - 예를 들어, $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$ 행렬은 각각 20x3, 3x10, 10x2라고 하면, $((\mathbf{A}_1 \times \mathbf{A}_2) \times \mathbf{A}_3)$은 600+400=1000번, $(\mathbf{A}_1 \times (\mathbf{A}_2 \times \mathbf{A}_3))$은 60+120=180번 곱셈. 두 번째가 5.5배 빠름
  - 모든 순서를 나열하면 경우의 수가 지수적이어서 계산 폭발
  - 동적 프로그래밍은 아주 효율적인 방법
  - $n-1$ 크기 문제의 답을 n 크기 문제의 답으로 확산하는 순환식

$$M[i,j] = \min_{k=i,i+1,\cdots,j-1} \left(M[i,k] + M[k+1,j] + d_{i-1} d_k d_j \right)$$

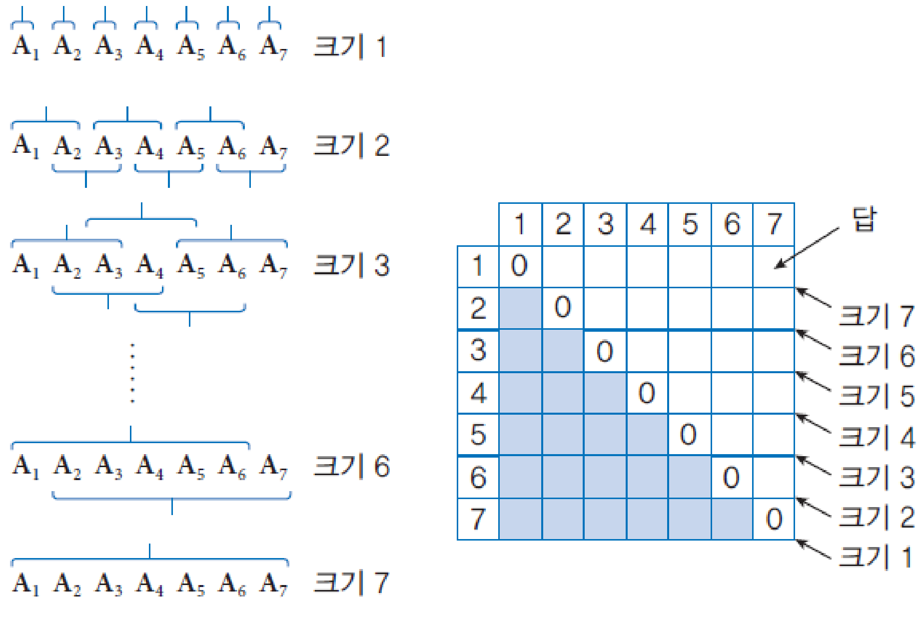

그림 3-2 행렬 곱의 최적 순서 정하기를 위한 동적 프로그래밍

3.1 원리

- 동적 프로그래밍을 강화 학습에 어떻게 적용할까?
  - 가장 작은 문제는 무엇이고 하나 더 큰 문제는 무엇일까? 작은 문제의 답을 하나 더 큰 문제로 파급하기 위한 순환식을 어떻게 유도하나?
    - FrozenLake에서는 H와 G로 표시한 종료 상태를 가장 작은 문제로 간주하면 합리적
    - [그림 3-3]에서 화살표는 점점 큰 문제로 향하는 과정
    - 에피소드 $[-, s_0]a_0[r_0, s_1]a_1 \cdots [r_{T-2}s_{T-1}]a_{T-1}[r_{T-1}, s_T]$에서 s_T는 크기 1인 문제, 일반적으로 s_i는 s_{i+1}보다 하나 더 큰 문제로 간주

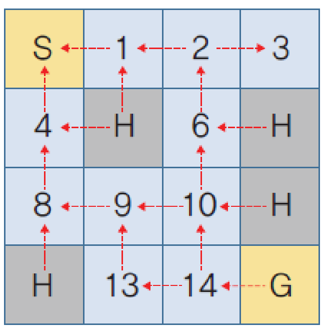

그림 3-3 FrozenLake 과업에 동적 프로그래밍 적용

- 강화 학습의 동적 프로그래밍
  - 알고리즘 분야에서는 크기 i와 $i+1$이 엄밀히 구분되지만 강화 학습에서는 불분명
  - 벨만 방정식은 이런 복잡성을 처리해 줌

3.2 벨만 방정식과 정책 반복 알고리즘

- 이 절은 벨만 방정식이라는 순환식을 유도
- 벨만 방정식은 모든 강화 학습 알고리즘의 근간을 형성
- 정확한 이해 필요

3.2.1 벨만 방정식

- 가치 함수를 위한 식 (2.10)을 변형하면,

$$
\begin{aligned}
v_\pi(s) &= \sum_{a\in\mathcal{A}(s)} \pi(a\,|\,s) q_\pi(s,a) \\
&= \sum_{a\in\mathcal{A}(s)} \pi(a\,|\,s)\left(r + \sum_{s'\text{에서 출발하는 모든 궤적 } z} p(z)R(z) \right) \\
&= \sum_{a\in\mathcal{A}(s)} \pi(a\,|\,s)\big(r + v_\pi(s')\big)
\end{aligned}
$$

- 벨만 방정식Bellman equation (또는 벨만 기대 방정식 Bellman expectation equation)

$$
v_\pi(s) = \sum_{a\in\mathcal{A}(s)} \pi(a\,|\,s)\big(r + \gamma v_\pi(s')\big), \forall s \in \mathcal{S} \tag{3.1}
$$

3.2.1 벨만 방정식

벨만 방정식의 동작 예시

- 상태 s=10에서 $v_\pi(s=10)$을 계산하는 사례

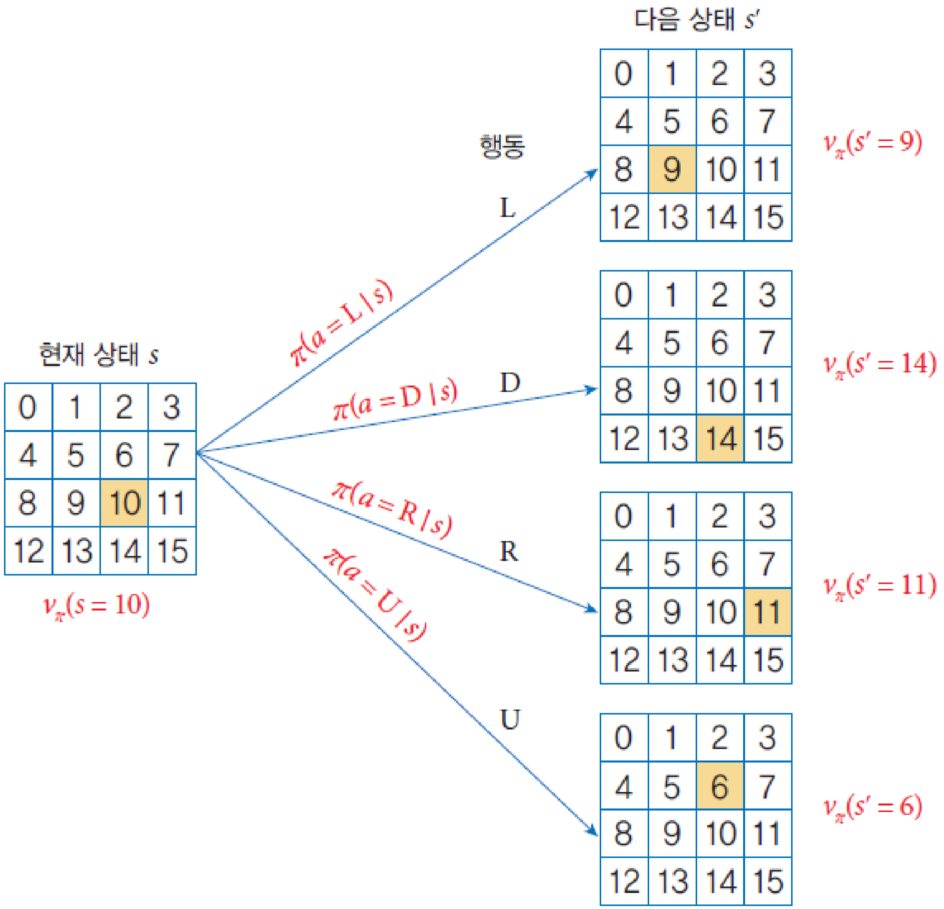

그림 3-4 결정론 MDP를 가진 FrozenLake 과업에서 벨만 방정식의 동작

3.2.1 벨만 방정식

- 가치 함수를 계산하는 방식의 비교
  - 벨만 방정식을 사용하는 [그림 3-5(a)]와 궤적을 나열하는 [그림 3-5(b)]
  - 식 (3.1)의 벨만 방정식을 이용: 다음 상태의 값 $v_\pi(s')$를 알면 현실적인 시간 내에 현재 상태의 값 $v_\pi(s)$를 계산할 수 있음
  - 궤적 나열하는 방식: 특수한 경우를 제외하고 궤적이 무한정 많아 계산 불가능함

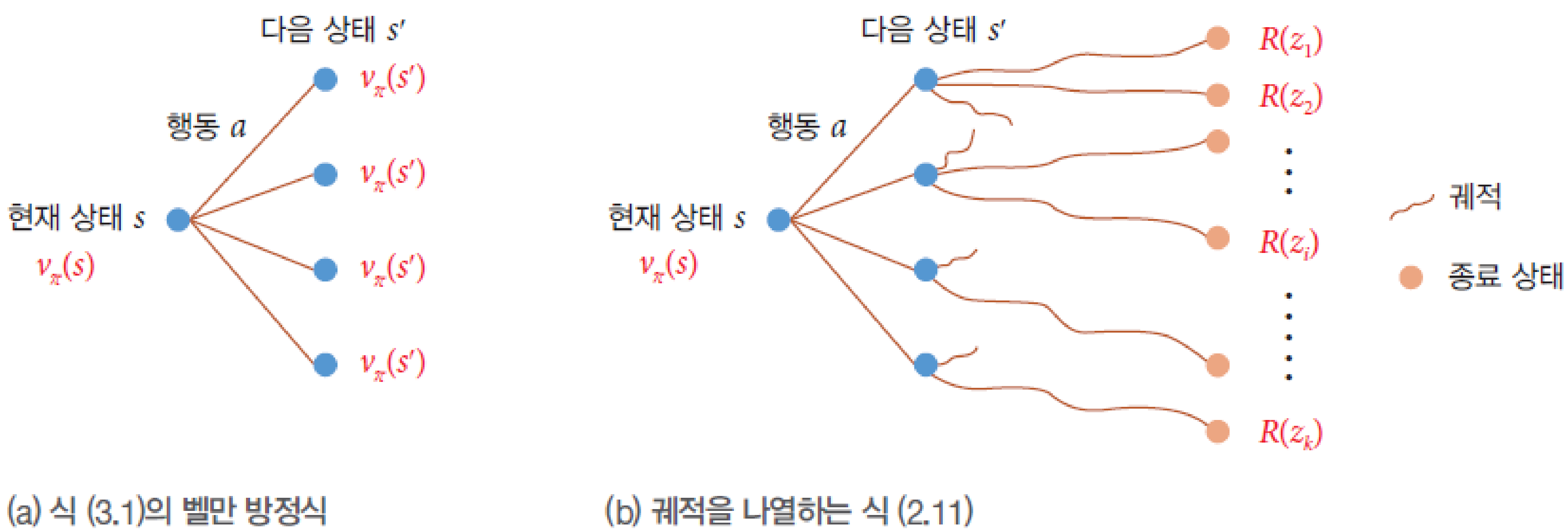

그림 3-5 가치 함수를 계산하는 공식의 동작 방식

3.2.1 벨만 방정식

- 벨만 방정식의 동작에 대해 생각해볼 점
  - 이웃 상태의 값 $v_\pi(s')$는 믿을 만한 값인가?
    - 학습 알고리즘은 가치 함수 값을 난수로 설정하고 출발하기 때문에 학습 도중에는 어떤 상태도 확정된 값을 가지지 않음
    - 벨만 방정식에서 상태는 이웃의 불완전한 값으로 자신을 개선. 동적 프로그래밍은 추정치를 가지고 추정하는 부트스트래핑bootstrapping 방식임
  - 수렴한 가치 함수는 자기 일관성self-consistency을 유지함
    - 어떤 상태에 대해 이웃 상태의 값을 가지고 벨만 방정식을 계산하면 자신의 값과 같음

3.2.2 정책 평가 알고리즘

- 벨만 방정식을 이용하여 정책 π를 평가하는 [알고리즘 3-1]

알고리즘 3-1 벨만 방정식을 이용한 정책 평가

입력: 정책 π

출력: 가치 함수 v_π (배열 V)

$|\mathcal{S}|$ 크기의 배열 V를 만든다.
모든 상태 s에 대해 $V(s)$를 0으로 초기화한다.
while TRUE
    $oldV = V$
    for $s \in \mathcal{S}$ // 상태 각각에 대해
        $V(s) = \Sigma_{a \in \mathcal{A}(s)} \pi(a|s)(r + \gamma V(s'))$ // 식 (3.1)의 벨만 방정식 적용
    if $\max_s |V(s) - oldV(s)| < \varepsilon$, break // 변화가 충분히 작으면 수렴으로 간주하고 루프 탈출
return V

3.2.3 프로그래밍 실습: 정책 평가

- [프로그램 3-1]은 FrozenLake에 대해 [알고리즘 3-1]을 구현
  - 13행의 env.unwrapped.P는 MDP의 전이 확률에 해당

프로그램 3-1 벨만 방정식을 이용하여 랜덤 정책을 평가(FrozenLake 과업)

```
1   import gymnasium as gym
2   import numpy as np
3
4   gamma=0.9 # 할인율
5
6   def policy_evaluation(env,policy):
7       V=np.zeros(env.observation_space.n) # 가치 함수를 저장할 표
8       while True:
9           oldV=V.copy()
10          for state in range(env.observation_space.n):
11              v=0
12              for action,action_prob in enumerate(policy[state]):  # 벨만 방정식
13                  prob,next_state,reward,terminated=env.unwrapped.P[state][action][0]
14                  v=v+action_prob*(reward+gamma*V[next_state])
15              V[state]=v
16          if max(np.abs(V-oldV))<1e-8:
```

3.2.3 프로그래밍 실습: 정책 평가

```
                break
        return V

env=gym.make('FrozenLake-v1',is_slippery=False,render_mode='ansi')

pi1=np.ones((env.observation_space.n,env.action_space.n))/env.action_space.n # 랜덤 정책
V=policy_evaluation(env,pi1)
print('pi1(랜덤 정책)의 가치 함수:\n',np.round(V.reshape([4,4]),4))
```

```
pi1(랜덤 정책)의 가치 함수:
 [[0.0045 0.0042 0.0101 0.0041]
 [0.0067 0.     0.0263 0.    ]
 [0.0187 0.0576 0.107  0.    ]
 [0.     0.1304 0.3915 0.    ]]
```

실행 결과 해석

- H와 G에 해당하는 종료 상태는 모두 가치 함수 값이 0
- 상태 14의 $v_{\pi_1}(s=14)=0.3915$. 상태 14에서 R을 선택하면 1의 보상을 받는데, 랜덤 정책인 π_1은 무작위로 네 방향으로 이동하므로 0.3915에 불과
- 상태 14에 대해 자기 일관성 확인
  - 1/4(0+0.9×0.1304)+1/4(0+0.9×0.3915)+1/4(1+0)+1/4(0+0.9×0.107)=0.3915

3.2.4 정책 반복 알고리즘

- [알고리즘 3-2]는 정책 반복 알고리즘
  - 정책을 평가하는 [알고리즘 3-1]을 최적 정책을 찾는 정책 반복 알고리즘으로 확장
    - 초기에는 좋은 정책에 대한 실마리가 없기 때문에 랜덤하게 정책을 초기화 (π_0)
    - π_0의 가치함수 v_{π_0}을 구하고, v_{π_0}을 이용하여 π_0를 π_1로 개선
    - 해당 과정을 수렴할 때까지 반복하여 최적 정책 π_*에 도달

3.2.4 정책 반복 알고리즘

■ 정책 반복 알고리즘의 개선 과정

- 식 (3.2)에서 E는 평가, I는 개선 단계

$$\pi_0 \xrightarrow{\mathrm{E}} v_{\pi_0} \xrightarrow{\mathrm{I}} \pi_1 \xrightarrow{\mathrm{E}} v_{\pi_1} \xrightarrow{\mathrm{I}} \pi_2 \xrightarrow{\mathrm{E}} v_{\pi_2} \cdots\cdots \xrightarrow{\mathrm{I}} \pi_* \tag{3.2}$$

- GPI(generalized policy iteration) 전략으로 설명하면,

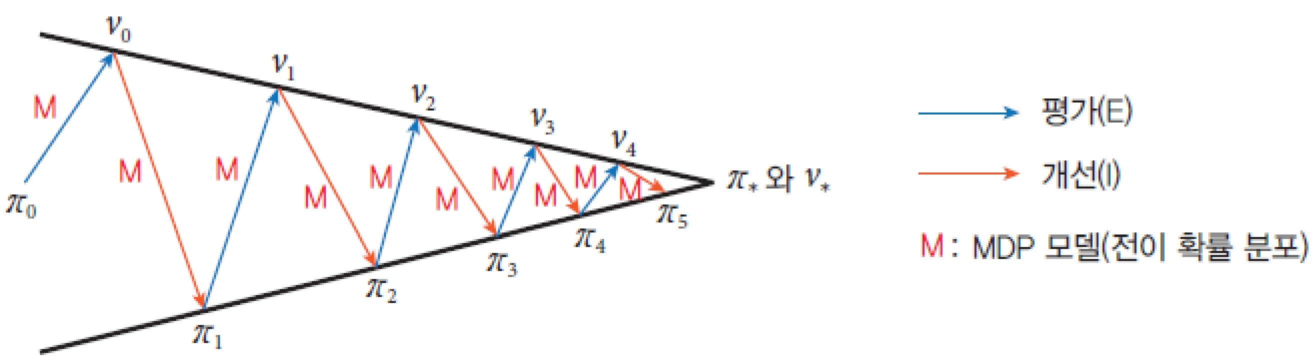

그림 3-6 정책 반복 알고리즘의 GPI 전략

- 정책을 평가하고 개선하는 과정에서 MDP 모델(전이 확률 분포)를 사용

3.2.4 정책 반복 알고리즘

- [알고리즘 3-2]는 정책 반복 알고리즘
  - 1행에서 정책을 난수로 설정하고 출발. E(정책 평가)와 I(정책 개선) 단계를 반복하면서 점점 최적 정책으로 접근
  - 7행에서 탐욕 선택. 따라서 정책을 2차원 표가 아니라 1차원 표에 기록

알고리즘 3-2 정책 반복 알고리즘

입력: 과업

출력: 최적 정책 π_*와 최적 가치 함수 v_*

1 정책 π를 난수로 초기화한다.
2 while True
3   [알고리즘 3-1]을 이용하여 가치 함수 v_π를 계산한다. // E(정책 평가) 단계
4   converged=True
5   for $s \in \mathcal{S}$ // I (정책 개선) 단계
6   old_action=$\pi(s)$
7         $\pi(s)=\mathrm{argmax}_a(r+\gamma v_\pi(s'))$
8   if old_action$\neq\pi(s)$, converged=False
9   if converged=True, break // 변화가 없으면 수렴으로 간주하고 루프 탈출
10 return π, v_π // π_*와 v_*를 반환

3.2.5 프로그래밍 실습: 정책 반복

- [프로그램 3-2]는 FrozenLake에 대해 [알고리즘 3-1]을 구현

프로그램 3-2 FrozenLake 과업을 위한 정책 반복 알고리즘

```
import gymnasium as gym
import numpy as np

gamma=0.9 # 할인율

def policy_iteration(env):
    V=np.zeros(env.observation_space.n)
    pi=[0 for i in range(env.observation_space.n)] # 정책을 행동 0으로 초기화
    while True:
        while True: # E(정책 평가)
            oldV=V.copy()
            for state in range(env.observation_space.n):
                action=pi[state]
                prob,next_state,reward,terminated=env.unwrapped.P[state][action][0]
                v=reward+gamma*V[next_state]
                V[state]=v
            if max(np.abs(V-oldV))<1e-8:
                break

        converged=True # I (정책 개선)
        for state in range(env.observation_space.n):
            old_action=pi[state]
            q=np.zeros(env.action_space.n)
            for action in range(env.action_space.n):
                prob,next_state,reward,terminated=env.unwrapped.P[state][action][0]
                q[action]=reward+gamma*V[next_state]
            pi[state]=np.argmax(q)
            if pi[state]!=old_action:
                converged=False
        if converged:
            break
    return V,pi
```

3.2.5 프로그래밍 실습: 정책 반복

```

env=gym.make('FrozenLake-v1',is_slippery=False,render_mode='ansi')

V,pi=policy_iteration(env)
print('최적 정책:\n',np.array(pi).reshape([4,4]))
print('최적 가치 함수:\n',np.round(V.reshape([4,4]),4))
```

```
최적 정책:
 [[1 2 1 0]
 [1 0 1 0]
 [2 1 1 0]
 [0 2 2 0]]
최적 가치 함수:
 [[0.5905 0.6561 0.729  0.6561]
 [0.6561 0.     0.81   0.    ]
 [0.729  0.81   0.9    0.    ]
 [0.     0.9    1.     0.    ]]
```

3.3 벨만 최적 방정식과 가치 반복 알고리즘

- 앞 절의 정책 반복 알고리즘은 정책 평가를 통해 정책을 개선하는 전략
- 이 전략은 계산량이 많은 심각한 문제
- 이 절에서는,
  - 정책 반복이 왜 계산량이 많은지 분석
  - 이를 개선한 가치 반복 알고리즘 소개
- 가치 반복은 벨만 방정식을 개조한 벨만 최적 방정식Bellman optimality equation 을 사용함

3.3.1 벨만 최적 방정식

- 정책 반복 알고리즘([알고리즘 3-2])의 분석
  - while(not converged) … while(not converged) … for each state …라는 3중 루프 형성
  - 계산량이 많은 이유
  - 식 (3.1)의 벨만 방정식을 개조하여 계산량을 줄일 필요성
- 식 (3.4)의 벨만 최적 방정식은 벨만 방정식에서 파생
  - $\sum_{a \in \mathcal{A}(s)} \pi(a|s)$의 기댓값 연산을 최댓값을 취하는 $\max_{a \in \mathcal{A}(s)}(.)$로 대체함
  - 식 (3.3)은 수렴을 마친 최적 가치 함수가 만족해야 하는 조건식

$$v_*(s) = \max_{a \in \mathcal{A}(s)} \left(r + \gamma v_*(s') \right) \quad (3.3)$$

  - 식 (3.3)에서 *를 제거하면 식 (3.4)의 벨만 최적 방정식

$$v(s) = \max_{a \in \mathcal{A}(s)} \left(r + \gamma v(s') \right) \quad (3.4)$$

3.3.2 가치 반복 알고리즘

식 (3.4)를 이용한 가치 반복 알고리즘

- 정책 없이 가치 함수만 개선함

$$v_0 \xrightarrow{\mathrm{I}} v_1 \xrightarrow{\mathrm{I}} v_2 \cdots\cdots \xrightarrow{\mathrm{I}} v_* \qquad (3.5)$$

- while(not converged)… for each state…라는 2중 루프를 형성하여 정책 평가보다 효율적

알고리즘 3-3 가치 반복 알고리즘(결정론 MDP)

입력: 과업

출력: 최적 정책 π_*와 최적 가치 함수 v_*

$|S|$ 크기의 배열 V를 만든다.
모든 상태 s에 대해 $V(s)$를 0으로 초기화한다.
while TRUE
    $oldV = V$
    for $s \in S$ // 상태 각각에 대해
        $V(s) = \max_a(r + \gamma V(s'))$ // 식 (3.4)의 벨만 최적 방정식 적용
    if $\max_s |V(s) - oldV(s)| < \varepsilon$, break // 변화가 충분히 작으면 수렴으로 간주
for $s \in S$ // 최적 가치 함수로부터 최적 정책 구하기
    $\pi(s) = \mathrm{argmax}_a(r + \gamma V(s'))$
return π, V // π_*와 v_*를 반환

3.3.3 프로그래밍 실습: 가치 반복

- [프로그램 3-3]은 FrozenLake에 대해 [알고리즘 3-3] 구현

프로그램 3-3 FrozenLake 과업을 위한 가치 반복 알고리즘

```
import gymnasium as gym
import numpy as np

gamma=0.9 # 할인율

def value_iteration(env):
    V=np.zeros(env.observation_space.n)
    while True: # 최적 가치 함수 추정
        oldV=V.copy()
        for state in range(env.observation_space.n):
            q=np.zeros(env.action_space.n)
            for action in range(env.action_space.n):
                prob,next_state,reward,terminated=env.unwrapped.P[state][action][0]
                q[action]=reward+gamma*V[next_state]
            V[state]=np.max(q)
        if max(np.abs(V-oldV))<1e-8:
            break

    pi=env.observation_space.n*[None] # 최적 가치 함수로부터 최적 정책 구하기
    for state in range(env.observation_space.n):
        q=np.zeros(env.action_space.n)
        for action in range(env.action_space.n):
            prob,next_state,reward,terminated=env.unwrapped.P[state][action][0]
            q[action]=reward+gamma*V[next_state]
        pi[state]=np.argmax(q)
    return pi,V
```

3.3.3 프로그래밍 실습: 가치 반복

```python

env=gym.make('FrozenLake-v1',is_slippery=False,render_mode='ansi')

pi,V=value_iteration(env)
print('최적 정책:\n',np.array(pi).reshape([4,4]))
print('최적 가치 함수:\n',np.round(V.reshape([4,4]),4))
```

```
최적 정책:
 [[1 2 1 0]
 [1 0 1 0]
 [2 1 1 0]
 [0 2 2 0]]
최적 가치 함수:
 [[0.5905 0.6561 0.729  0.6561]
 [0.6561 0.     0.81   0.    ]
 [0.729  0.81   0.9    0.    ]
 [0.     0.9    1.     0.    ]]
```

[프로그램 3-2]와 결과가 같음

3.4 스토캐스틱 과업과 동적 프로그래밍

- 앞 절의 벨만 방정식과 벨만 최적 방정식은 결정론 MDP에 국한
- 결정론 MDP는 실제 세계를 단순화한 측면
  - 예를 들어, 로봇의 경우 바람, 지형, 조명 등의 외부 환경을 고려하면 결정론 만족 못함
- 이런 불확실성을 반영하려면 스토캐스틱 MDP 필요

3.4.1 벨만 최적 방정식

- 식 (3.6)은 스토캐스틱 환경을 위한 벨만 방정식
  - 식 (3.1)에 $\sum_{s'}\sum_{r} p(s',r|s,a)$ 항이 추가된 꼴. 이 항이 스토캐스틱 성질 반영

$$v_\pi(s) = \sum_{a\in\mathcal{A}(s)} \underbrace{\pi(a|s)}_{\text{정책}} \sum_{s'}\sum_{r} \underbrace{p(s',r|s,a)}_{\text{MDP 전이 확률}} \underbrace{\left(r+\gamma v_\pi(s')\right)}_{\text{이득}} \quad (3.6)$$

- 식 (3.7)은 스토캐스틱 환경을 위한 벨만 최적 방정식

$$v(s) = \max_{a\in\mathcal{A}(s)} \sum_{s'}\sum_{r} p(s',r|s,a)\left(r+\gamma v(s')\right) \quad (3.7)$$

3.4.1 벨만 최적 방정식

벨만 방정식을 위한 동작 예시

- 상태 s=10에서 $v_\pi(s = 10)$을 계산하는 사례

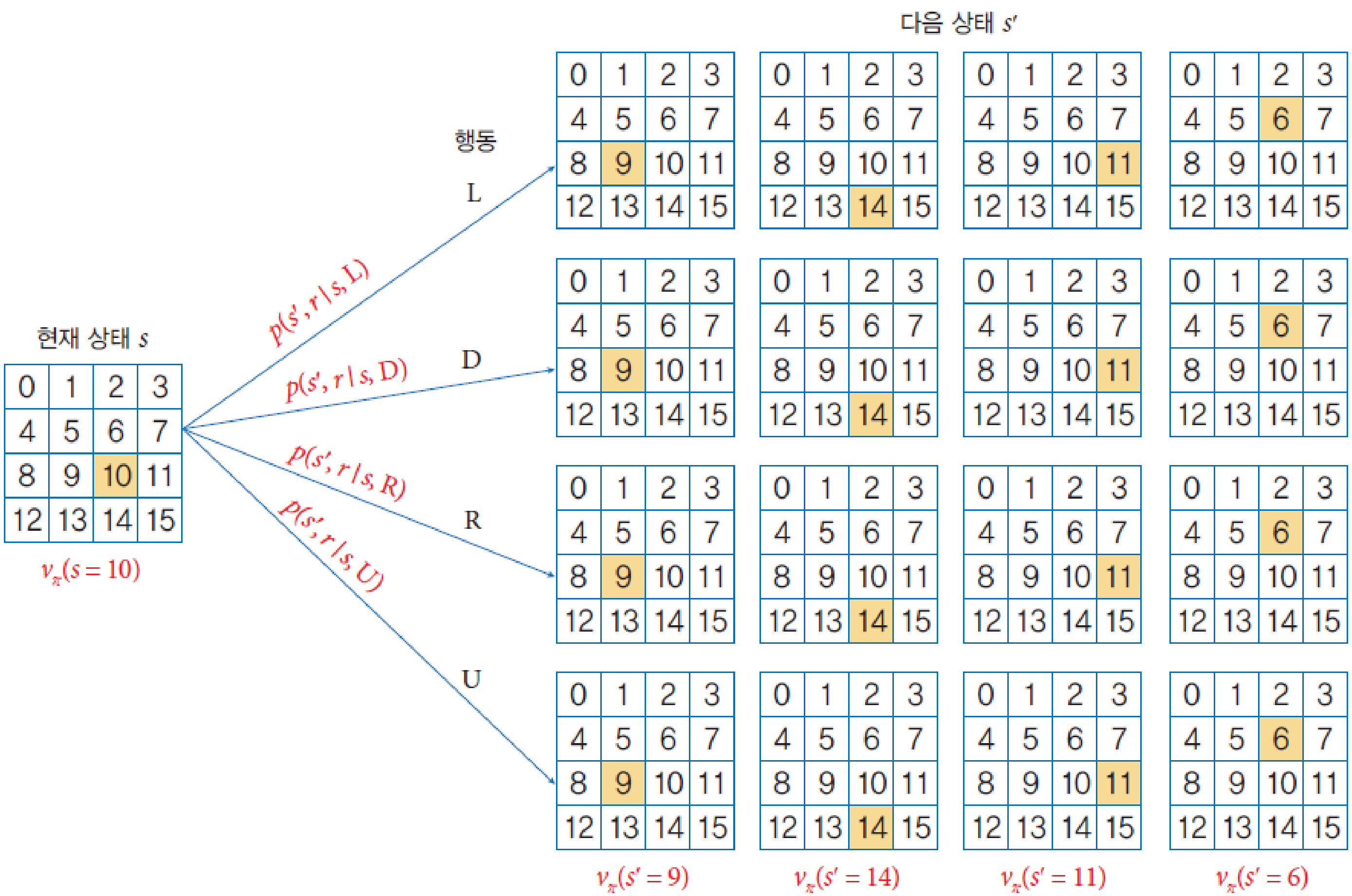

그림 3-7 스토캐스틱 MDP를 가진 FrozenLake 과업에서 벨만 방정식의 동작

3.4.2 가치 반복 알고리즘

- [알고리즘 3-4]는 스토캐스틱 MDP를 위한 가치 반복 알고리즘
  - [알고리즘 3-3]과 기본 절차는 같은데 6행과 9행을 식 (3.7)로 바꾼 점만 다름

알고리즘 3-4 가치 반복 알고리즘(스토캐스틱 MDP)

입력: 과업

출력: 최적 정책 π_*와 최적 가치 함수 v_*

```
1   |𝒮| 크기의 배열 V를 만든다.
2   모든 상태 s에 대해 V(s)를 0으로 초기화한다.
3   while TRUE
4       oldV=V
5       for s∈𝒮    // 상태 각각에 대해
6           V(s)=max_a Σ_s' Σ_r p(s',r|s,a)(r+γV(s'))    // 식 (3.7)의 벨만 최적 방정식 적용
7       if max_s|V(s)-oldV(s)|<ε, break    // 변화가 충분히 작으면 수렴으로 간주
8   for s∈𝒮    // 최적 가치 함수로부터 최적 정책 구하기
9       π(s)=argmax_a Σ_s' Σ_r p(s',r|s,a)(r+γ(s'))
10  return π, V    // π_*와 v_*를 반환
```

3.4.3 프로그래밍 실습: 가치 반복

스토캐스틱 MDP의 전이 확률에 대한 이해

- FrozenLake 과업에서 실험

프로그램 3-4 FrozenLake 과업의 MDP 전이 확률 분포를 확인

```
import gymnasium as gym
import pprint # 딕셔너리를 깔끔하게 출력

env=gym.make('FrozenLake-v1',is_slippery=False,render_mode='ansi')
print('결정론 MDP의 전이 확률 분포:')
pprint.pprint(env.unwrapped.P)

env=gym.make('FrozenLake-v1',is_slippery=True,render_mode='ansi')
print('스토캐스틱 MDP의 전이 확률 분포:')
pprint.pprint(env.unwrapped.P)
```

3.4.3 프로그래밍 실습: 가치 반복

- 결정론에서는 상태 0에서 행동 0을 취하면 확률 1로 상태 0으로 전이
- 스토캐스틱에서는 상태 0에서 행동 0을 취하면 확률 0.333으로 상태 0, 확률 0.333으로 상태 0, 확률 0.333으로 상태 4로 전이

```
결정론 MDP의 전이 확률 분포:
{0: {0: [(1.0, 0, 0.0, False)],
     1: [(1.0, 4, 0.0, False)],
     2: [(1.0, 1, 0.0, False)],
     3: [(1.0, 0, 0.0, False)]},
…
```

```
스토캐스틱 MDP의 전이 확률 분포:
{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)]},
…
```

3.4.3 프로그래밍 실습: 가치 반복

프로그램 3-5 FrozenLake 과업을 위한 가치 반복 알고리즘(스토캐스틱 환경)

```
import gymnasium as gym
import numpy as np

gamma=0.9 # 할인율

def value_iteration(env):
    V=np.zeros(env.observation_space.n)
    while True: # 최적 가치 함수 추정
        oldV=V.copy()
        for state in range(env.observation_space.n):
            q=np.zeros(env.action_space.n)
            for action in range(env.action_space.n):
                for prob,next_state,reward,terminated in env.unwrapped.P[state][action]:
                    q[action]=q[action]+prob*(reward+gamma*V[next_state])
            V[state]=np.max(q)
        if max(np.abs(V-oldV))<1e-8:
            break

    pi=env.observation_space.n*[None] # 최적 가치 함수로부터 최적 정책 구하기
    for state in range(env.observation_space.n):
        q=np.zeros(env.action_space.n)
        for action in range(env.action_space.n):
            for prob,next_state,reward,terminated in env.unwrapped.P[state][action]:
                q[action]=q[action]+prob*(reward+gamma*V[next_state])
        pi[state]=np.argmax(q)
    return pi,V
```

$\sum_{s'}\sum_{r} p(s',r|s,a)$을 구현

3.4.3 프로그래밍 실습: 가치 반복

- 실망스런 결과 얻음(같은 확률로 세 방향 전이하기 때문)

```
env=gym.make('FrozenLake-v1',is_slippery=True,render_mode='ansi')

pi,V=value_iteration(env)
print('최적 정책:\n',np.array(pi).reshape([4,4]))
print('최적 가치 함수:\n',np.round(V.reshape([4,4]),4))
```

```
최적 정책:
 [[0 3 0 3]
 [0 0 0 0]
 [3 1 0 0]
 [0 2 1 0]]
최적 가치 함수:
 [[0.0689 0.0614 0.0744 0.0558]
 [0.0919 0.     0.1122 0.    ]
 [0.1454 0.2475 0.2996 0.    ]
 [0.     0.3799 0.639  0.    ]]
```

3.4.3 프로그래밍 실습: 가치 반복

■ 의도한 방향 확률을 높인 MDP에서는 제대로 작동

프로그램 3-6 FrozenLake 과업을 위한 가치 반복 알고리즘(의도한 방향의 확률을 높인 스토캐스틱 환경)

아래 코드를 제외하고 [프로그램 3-5]와 같음

```
…
env=gym.make('FrozenLake-v1',is_slippery=True,render_mode='ansi')

prob=[0.1,0.8,0.1] # 전이 확률 (1/3,1/3,1/3)을 (0.1,0.8,0.1)로 수정
for state in range(env.observation_space.n):
    for action in range(env.action_space.n):
        if len(env.P[state][action])==3:
            for i,transition in enumerate(env.unwrapped.P[state][action]):
                env.P[state][action][i]=(prob[i],transition[1],transition[2],
                transition[3])

pi,V=value_iteration(env)
print('최적 정책:\n',np.array(pi).reshape([4,4]))
print('최적 가치 함수:\n',np.round(V.reshape([4,4]),4))
```

```
최적 정책:
 [[1 2 1 0]
 [1 0 1 0]
 [2 1 1 0]
 [0 2 2 0]]
최적 가치 함수:
 [[0.3804 0.3589 0.4536 0.3589]
 [0.436  0.     0.5403 0.    ]
 [0.551  0.7108 0.7504 0.    ]
 [0.     0.8246 0.9533 0.    ]]
```

3.5 동적 프로그래밍의 특성과 한계

- 동적 프로그래밍은 초창기 알고리즘
  - 현대적 관점으로 한계. 현대에는 잘 쓰이지 않음
  - 하지만 중요한 개념을 많이 내포하고 있어 공부할 가치 있음
- 동적 프로그래밍은 모델 기반 알고리즘
  - 3장은 MDP 확률을 사용하는 원시적인 모델 기반

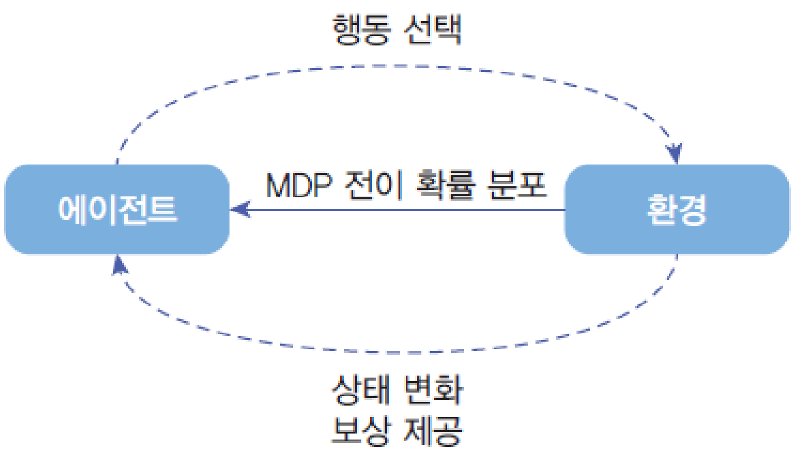

그림 3-8 모델 기반 알고리즘

  - 현대에는 모델을 학습하는 발전된 형태의 모델 기반(13.3.4절의 뮤제로)
  - 본 강의에서는 3장을 제외한 4~11장은 모델 자유

3.5 동적 프로그래밍의 특성과 한계

- 동적 프로그래밍은 참조표lookup table 기반
  - 정책은 $|\mathcal{S}| \times |\mathcal{A}|$, 상태 가치 함수는 $|\mathcal{S}|$, 행동 가치 함수는 $|\mathcal{S}| \times |\mathcal{A}|$ 크기의 배열 사용
  - 상태 공간의 크기 $|\mathcal{S}|$가 큰 과업에서는 메모리가 기하급수적으로 커지는 문제
  - 상태 공간이 연속인 경우 이산화 과정 필요. 비효율적이고 부자연스러움
- 상태 공간을 빠짐없이 모두 탐색
  - [알고리즘 3-3]의 5행과 8행의 for $s \in \mathcal{S}$에서 수행
  - $|\mathcal{S}|$가 큰 과업에서는 계산량이 기하급수적으로 커지는 문제
- 부트스트래핑 방식
  - 이웃과 값을 주고 받으면서 값을 개선해 나가는 방식
  - 추정치를 통해 추정하는 방식

Thank you

Jahoon Koo
(sigmao@korea.ac.kr)