4. Dynamic Programming

dynamic programming이라는 용어는 알고리즘의 집합이다. MDP 환경에서 완벽한 모델이 주어졌을 때 optimal policies를 계산할 때 쓰는 알고리즘들.
고전적 DP 알고리즘은 강화학습에서 용도가 제한적이었어.
완벽한 모델에 대한 가정과 computational expense 때문이었지.
하지만 이론적으론 여전히 중요.
책의 나머지에서 볼 methods들도 사실 위 두 가지를 극복하면서 DP랑 최대한 같은 효과를 내려고 하는 시도들로 볼 수 있어.

이 장부터, 환경은 주로 finite MDP로 가정한다.
그러니까, state, action, reward sets 가 finite 이고,
dynamics가

$$p(s', r \mid s, a), \text{ for all } s \in \mathcal{S},\ a \in \mathcal{A}(s),\ r \in \mathcal{R}, \text{ and } s' \in \mathcal{S}^+$$

의 확률 set에서 주어진다는 뜻.
DP ideas가 continuous state and action spaces 문제에도 적용될 수 있지만,
exact solutions은 special cases 에서만 가능.
continuous states and actions 문제의 approximate solutions을 얻기 위한 common way는,
quantize 해서 finite-state DP methods에 적용하는 것.
Ch 9 에서 continuous prob. 에 적용되는 methods들을 볼 거야.

DP의 key idea는, good policies 탐색을 organize 하고 structure 하는 value functions을 사용하는 거야.
이 챕터에서 Ch 3 에서 정의된 value functions을 계산하는 데 DP를 어떻게 쓰는지 보자.
앞에서 논의했던대로, optimal policies는 쉽게 찾을 수 있어.
optimal value functions v_*, q_* 를 찾기만 하면.
걔들은 Bellman optimality equations을 만족하지.

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_*(s')\Big] \end{aligned} \tag{4.1}$$

or

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\Big|\; S_t = s, A_t = a\Big] \\ &= \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big], \end{aligned} \tag{4.2}$$

DP 알고리즘은 이런 Bellman equations을 assignments로 바꿔서 얻는데,
그러니까, 원하는 value functions을 approximations 하는 걸 improving 하는 update rules 같은 assignments로 바꾸는거지.

4.1 Policy Evaluation(Prediction)

먼저 arbitrary policy π 의 state-value function v_π 를 계산하는 방법을 알아보자.
이걸DP에선 policy evaluation 라고 하지.

prediction problem 이라고도 불러.
Ch 3에서 이거 기억나나?

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] && \text{(from (3.9))} \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] && (4.3) \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma v_\pi(s') \Big], && (4.4)
\end{aligned}
$$

π(a|s) 는 policy π에서 state s일 때 action a를 하는 확률.
π가 subscripted 된 expectations 는 뒤따르는 π에 conditional 하다는 말.
v_π 의 existence와 uniqueness는
γ < 1 이거나
under π의 모든 states에서 eventual termination 이 보장될 때
보장된다.
environment의 dynamics를 다 알고 있으면 (4.4)는 |S| 의.. 이건 뭔 말이지.
원칙적으로, 이것의 solution은 straightforward하다. 계산만 좀 거시기할 뿐.
우리의 목적을 위해, iterative solution methods가 가장 suitable 하다.
approximate value functions 의 sequence v_0, v_1, v_2, …, 를 생각해보자.
얘들은 각각 S^+ 에서 R로 가는 mapping이다.
initial approximation v_0는 임의로 골랐다.(terminal state가 있으면 value 0 로 주어져야 함)
각 연속적인 approximation은 (4.4)의 Bellman equation for v_π 를 update rule로 해서 구함.

$$
\begin{aligned}
v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma v_k(s') \Big], && (4.5)
\end{aligned}
$$

S에 속한 모든 s에 대해서.

$v_k = v_\pi$ 는 fixed point 다. 이 update rule에서.
왜냐면 이 경우에 v_π 의 Bellman equation은 equality 를 보장하기 때문.
sequence $\{v_k\}$ 는 일반적으로 k가 무한대로 가면 v_π로 converge함을 보일 수 있다. v_π 의 existence를 보장하는 조건과 같은 조건일 때.
이 algorithm을 iterative policy evaluation이라고 한다.

v_k에서 v_{k+1}을 생성하는, each successive approximation을 produce하기 위해
iterative policy evaluation은 각 state s에 같은 operation을 적용한다.
s의 old value를
under the policy being evaluated에서 가능한 모든 one-step transitions에 따르는
successor states of s의 old values,
expected immediate rewards
에서 얻은 새로운 value로 바꾼다.
이런 operation을 expected update라고 부르는겨.

iterative policy evaluation의 각 iteration은 모든 state의 value를 한 번 update해서 새로운 approximate value function v_{k+1} 를 만든다.
여러 다른 expected updates가 있는데,
state 나 state-action pair 가 update 되고 있는지에 따라,

또는, successor states의 estimated value들이 합쳐지는 정확한 방법에 따라 종류가 달라.

모든 DP알고리즘에서의 updates는 'expected' updates라고 불러.
왜냐면, next state의 sample이 아니라 모든 가능한 next state의 expectation에 기초하고 있거든.

update의 nature는 equation으로 표현 가능. 위처럼.
아니면 backup diagram으로도 가능.
iterative policy evaluation에서 쓰인 expected update는 47 page의 backup diagram과 대응.

(4.5)에 주어진 iterative policy evaluation 을 적용하는 sequential computer program 을 짜려면, two arrays을 이용해야 할 거야.
하나는 old values v_k 를 위한 것,
하나는 new values v_{k+1} 을 위한 것.
이 array 두 개로, new values 가 하나씩 계산됨. old values로부터. old values 바꾸지 않으면서.

한 개 array를 쓰면서 old value들을 new value로 덮어씌우는 게 더 쉽긴 함.
states가 update 되는 순서에 따라 old ones 대신 new values가 쓰일 수 있어.
(4.5)의 오른쪽에 있는 애들에 말이지.
이 in-place algorithm도 v_π에 converge.
사실, 이게 two-array 보다 converge가 빨라.
왜냐면, new data가 가능하면 바로바로 쓰거든.
updates가 state space를 sweep 하면서 이루어진다고 보면 된다?

in-place 알고리즘에선, sweep 중에 어떤 states의 values가 update되는지 순서가 중요해.
rate of convergence에 크게 영향을 미쳐.
DP algorithms을 생각할 땐 보통 in-place 버젼을 떠올린다.

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

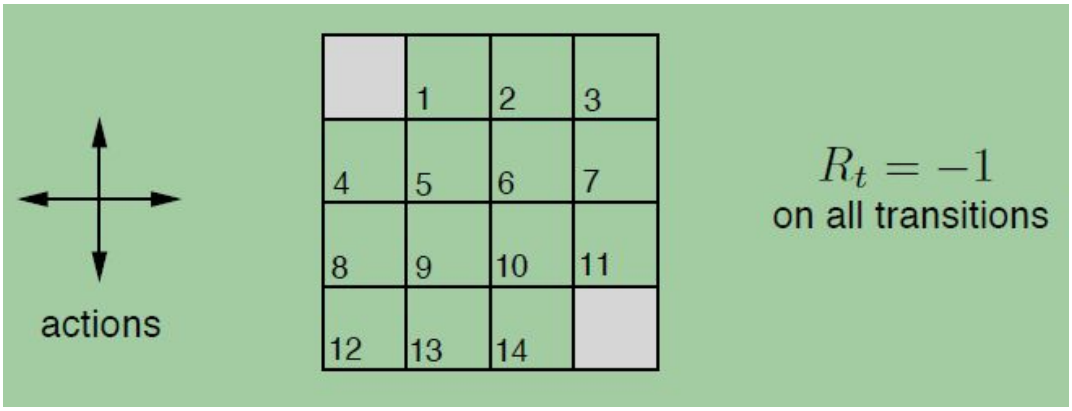
Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

이건 iterative policy evaluation의 complete in-place version을 나타낸 pseudocode 이다.
termination을 어떻게 처리하는지 봐.
Formally, iterative policy evaluation 은 limit에서만 converge.
하지만 in practice에선 그 전에 halt 돼야한다?

pseudocode는 quantity $\max_{s \in \mathcal{S}} |v_{k+1}(s) - v_k(s)|$ 를 test 해. 매 sweep과 stops 하고 나서 마다. sufficiently small 할 때.

(Example 4.1) 4x4 gridworld를 보자.



그림만 봐도 뭘 하려는지 감이 오쥬?
nonterminal states S = {1,2,...,14}
A = {up, down, right, left}, state transition에 deterministically 대응. 밖으로 나가는 actions은 state 안 바뀌고 그대로.
예를 들어,
p(6, -1 | 5, right) = 1
p(7, -1 | 7, right) = 1
p(10, r | 5, right) = 0 for all r ∈ R.
이건 undiscounted, episodic task 야.
teminal state에 닿기까지 모든 transitions의 reward는 -1.
terminal state는 같은 색깔. 두 군데지만 formally one state.
따라서, expected reward function은 r(s,a,s') = -1 for all states s, s' and actions a
agent가 모든 actions 에 대한 확률이 동일한 equiprobable random policy를 따른다고 해보자.
아래 (fig4.1) 은 iterative policy evaluation에 따라 계산된 value functions $\{v_k\}$ 의 sequence이다.
final estimate v_π 는, 이번 경우에, 각 state에서 termination까지의 expected steps 수를 무시한다.

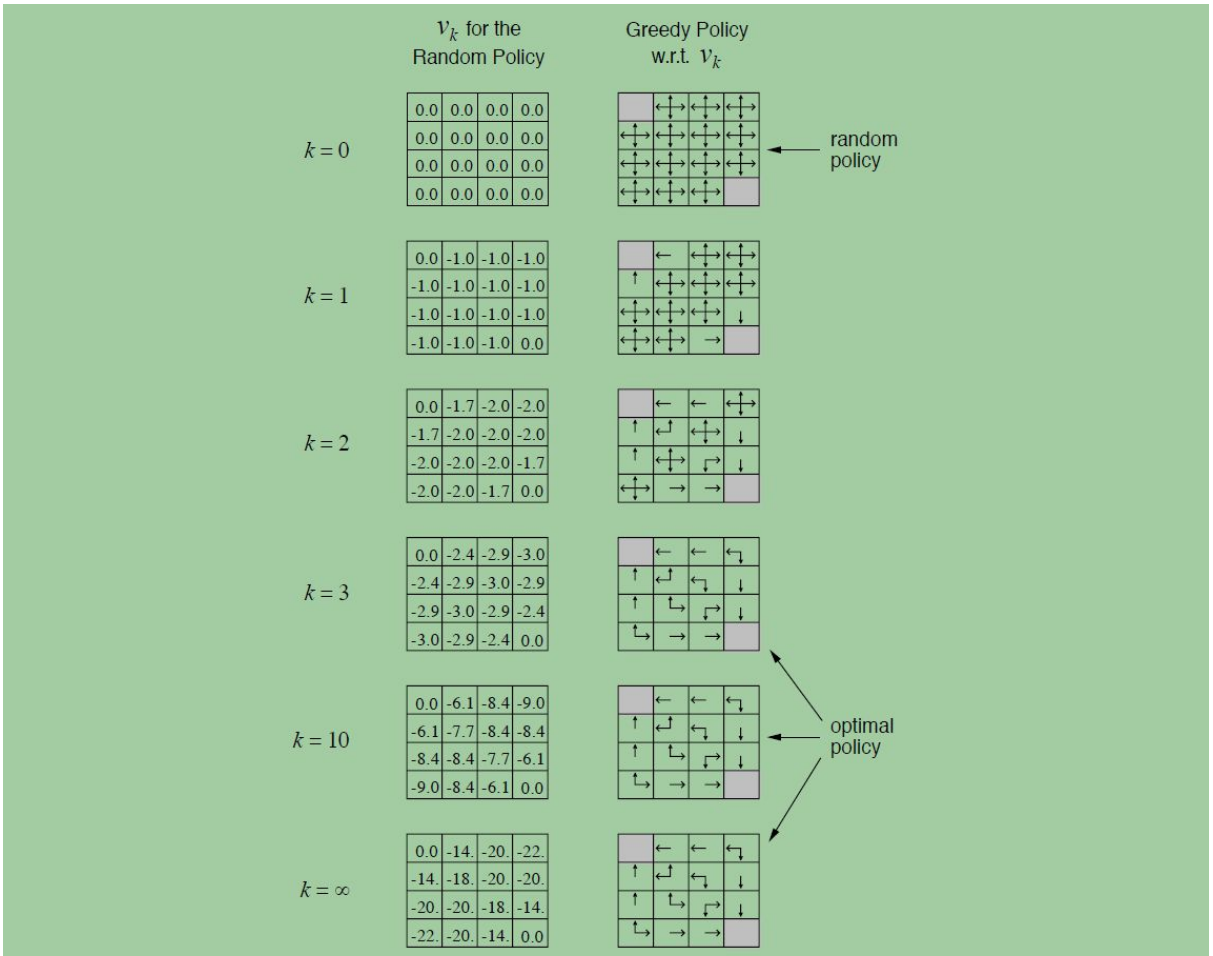


Figure 4.1: Convergence of iterative policy evaluation on a small gridworld. The left column is the sequence of approximations of the state-value function for the random policy (all actions equal). The right column is the sequence of greedy policies corresponding to the value function estimates (arrows are shown for all actions achieving the maximum). The last policy is guaranteed only to be an improvement over the random policy, but in this case it, and all policies after the third iteration, are optimal.

(fig 4.1) iterative policy evalutation 의 convergence다.
왼쪽: random policy일 때 state-value function의 approximation sequence.
오른쪽: value function estimates 에 대응하는 greedy policies의 sequence. maximum 을 달성하는 화살표다. 마지막 policy는 random policy의 개선으로만 보장되는데,
이 경우는 3번째 iteration 이후의 모든 policy가 다 optimal이다.

(Exercise4.1) Example 4.1 에서, π 가 equiprobable random policy, 그러니까 모든 actions 을 취할 확률이 다 같은 policy면, q_π(11, down)은 뭐니? q_π(7, down)은?

(Exercise4.2)
(Exercise4.3)

4.2 Policy Improvement
우리가 policy의 value function을 계산하는 이유는 better policy를 찾기 위해서야.
임의의 deterministic policy π에 대해 v_π 를 구했다 쳐봐.

어떤 state s에서 policy를 deterministically하게 action a ≠ π(s) 를 선택하도록 바꿔야하는지 알고 싶어.
우린 s에서 지금의 policy를 따르는 게 얼마나 good인지 알고 있어. 그게 v_π(s) 지.
하지만 새로운 policy로 바꾸는 게 좋을까 나쁠까?
이 물음에 답하는 한 가지 방법은, s에서 a를 고르고, 그런 다음 existing policy π 를 따르는거야.
이런 방식의 행동의 value는

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big]. \end{aligned} \tag{4.6}$$

이것.

key criterion은 이게 v_π보다 크냐 작냐야.
크면, 그러니까 항상 π를 따르는 것보다 s에서 a를 한 번 선택하고 π를 따르는 게 낫다면, s일 때마다 a를 선택하는 게 낫다고 기대할 수 있어.
그리고 새로운 policy가 전체적으로 더 나은 policy가 될 거야.

이게 사실이라는 건 general result의 special case 이고 policy improvement theorem이라고 불러.
π와 π' 이 모든 s에서
$q_\pi(s, \pi'(s)) \geq v_\pi$ (4.7)
인 deterministic policies 의 pair라고 하자.
그럼 policy π' 는 π 만큼, 혹은 더 좋아.
그러니까, π'은 모든 s에서 같거나 큰 expected return을 얻어.
$v_{\pi'}(s) \geq v_\pi$ (4.8)
그게 이 말인듯.
여기에 더해서, (4.7)에서, 모든 state에서 strict inequality 면, (4.8)에서 적어도 한 state에서 strict inequality 가 성립.
이 결과는 특히 두 policies에 적용된다. original deterministic policy π 와 changed policy π'.
π'는 π 랑 같은데
$\pi'(s) = a \neq \pi(s)$ 만 달라.
당연히? (4.7)은 s 외의 모든 state에서 만족.
그러니, $q_\pi(s,a) > v_\pi(s)$ 이면, 바뀐 policy가 π 보다 낫다는 거야.

policy improvement theorem 의 증명 idea 는 이해하기 쉽대.
(4.7)에서 시작해서, q_π side를 (4.6)으로 확장하고, (4.7)을 $v_{\pi'}(s)$ 를 얻을 때까지 다시 적용하면 돼.

$$
\begin{aligned}
v_\pi(s) &\le q_\pi(s, \pi'(s)) \\
&= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = \pi'(s)] && \text{(by (4.6))} \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\
&\le \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2})] \mid S_t = s] \\
&= \mathbb{E}_{\pi'}\big[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) \;\big|\; S_t = s\big] \\
&\le \mathbb{E}_{\pi'}\big[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) \;\big|\; S_t = s\big] \\
&\;\;\vdots \\
&\le \mathbb{E}_{\pi'}\big[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \;\big|\; S_t = s\big] \\
&= v_{\pi'}(s).
\end{aligned}
$$

한 번 적어보자.

지금까지 우리는, 주어진 policy와 policy의 value function에서, single state에서 특정 action을 하는 policy의 변화를 어떻게 평가하는지 봤다.
매 state에서 $q_\pi(s,a)$에 따라 best인 action을 고르는, 모든 states에서 모든 가능한 actions 의 changes를 고려하는 건 아주 내츄럴한 확장이다.
그러니까, 새로운 greedy policy π'를 고려하는 거. π'는

$$
\begin{aligned}
\pi'(s) &\doteq \arg\max_a q_\pi(s,a) \\
&= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] && (4.9) \\
&= \arg\max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_\pi(s')\Big],
\end{aligned}
$$

로 주어지는 거.
greedy policy는 v_π 에 따라 단기적(one step 이후)으로 best로 보이는 action을 취한다.
구조에 의해, greedy policy는 policy improvement theorem(4.7)의 조건을 만족한다.
그래서 original policy에 비해 좋은지 안다.
policy improvement: original policy의 value function에 대해 greedy하게 만듦으로써, original policy에서 향상된 새로운 policy를 만드는 process.

새로운 greedy policy를 π'가 old policy π 만큼 좋다고 하자. 더 나은 건 아니고.
그럼 $v_\pi = v_{\pi'}$ 이고, (4.9)로부터 모든 s에 대해

$$
\begin{aligned}
v_{\pi'}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_{\pi'}(s')\Big].
\end{aligned}
$$

하지만 이건 Bellman optimality equation (4.1) 과 같아.
그러니, v_π 는 v_* 일 거고(must be), π 와 π'는 둘 다 optimal policy야.
그래서 Policy improvement는 original policy가 이미 optimal 이 아닌 이상 반드시 더 좋은 policy를 준다.

여기까지 deterministic policy의 special case를 살펴봤다.
general case에선, stochastic policy π가 각 s에서 a를 취할 확률을 $\pi(a|s)$ 로 specify 한다.
detail은 안하겠지만 이 section의 모든 ideas는 stochastic policies로 확장 가능. 쉽게.
게다가, (4.9)에서처럼? policy improvement에서 tie 가 있으면,

그러니까 maximum을 얻는 여러 actions이 있으면,
stochastic case에서는 걔들 중에서 single action을 고를 필요가 없다.
대신, 새로운 greedy policy 에서 선택될 확률의 비율로 maximizing actions을 얻을 수 있다.
submaximal(maximal이 아닌 애들) actions 확률이 0으로 주어지는 한,
어떤 apportioning scheme도 허용된다.

(Fig4.1)의 마지막 줄은 stochastic policy에서 policy improvement의 예다.
여기서 original policy π 는 equiprobable random policy.
그리고 new policy π' 는 v_π 에 대해 greedy.
value function v_π 는 bottom-left 에 있는 diagram 이고,
set of possible π'가 bottom-right 에 있는 diagram.
π' diagram에 있는 multiple arrow 가 있는 states는 (4.9) 에서 maximum을 달성하는 여러 actions 임. 이 actions들 중 확률을 어떻게 분배하든 상관없음.
$v_{\pi'}$(s) 는 inspection에 의해, 모든 state에서 -1, -2 or -3으로 볼 수 있다.
반면 v_π(s)는 많아봐야 -14.
그러니 모든 states에서 $v_{\pi'}$(s) ≥ v_π(s) 이고 policy improvement를 나타내고 있다.
이 case에서 new policy π'는 optimal이었지만, 일반적으론 improvement만 보장된다.

4.3 Policy Iteration
v_π로 policy π가 π'로 improve되면 $v_{\pi'}$를 계산할 수 있다.
그러면 π''로 또 improve할 수 있지.
그러니까 monotically improving 하는 policies 와 value functions의 sequence를 얻을 수 있다는 거야.

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

E는 policy evaluation, I 는 policy improvement를 나타내.
각 policy는 이전의 것보다 strict improvement를 보장.(이미 optimal인 경우만 아니면.)
왜냐하면 finite MDP는 finite number of policies만 고려하고,
이 process는 finite number of iterations 안에 반드시 optimal policy와 value function에 수렴해야하기 때문이지.
이렇게 optimal policy를 찾는 방식을 policy iteration이라 함.
전체 알고리즘은 아래.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

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

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
      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

매 policy evaluation(iterative computation) 은 이전 policy의 value function로 시작한다.
이게 policy evaluation의 convergence speed를 엄청 높여준다고?
아마 value function이 다음으로 넘어갈 때 조금만 바뀌기 때문.

Policy iteration 은 iteration을 조금만 돌아도 often 수렴한다. 놀라움.
(fig4.1)에서 표현돼 있지.
bottom-left diagram은 value function인데 equiprobable random policy의 v지.
bottom-right diagram은 이 value function의 greedy policy 이고.
하지만, 이 경우에 policies 들은 그냥 better가 아니라 optimal임. 최소한의 스텝으로 terminal states로 proceeding하는.
이 예에서는, policy iteration이 1 iteration 만으로 optimal policy 를 찾을 거임.

(Exercise 4.4) 이 페이지의 policy iteration은 조그만 버그가 있는데, 2개 혹은 그 이상의 policies가 equally good이어서 계속 서로 바뀌면 terminate 되지 않는다는 것.
이건 pedagogy론 ok. but, actual use에선 아니겠지.
pseudocode를 수정해보렴. convergence가 보장되게.

(Example 4.2) Jack's Car Rental
Jack이 자동차 렌탈 샵을 두 개 운영하고 있다네.
Jack has a car available이면 10$ 주고 렌탈?
해당 location에 차가 없으면 거래는 잃는거지.
자동차는 반납 다음날부터 렌탈 가능.
자동차가 필요한 곳에 분배하기 위해 Jack은 두 location을 왔다갔다하는데, 한 대당 2$ cost.
각 location에서 requested, returned cars 숫자는 Poisson random variables 라 가정.

그러니까, cars 숫자 n 의 확률은 $\frac{\lambda^n}{n!}e^{-\lambda}$, λ 는 expected number.
requests 의 λ 가 3, 4.
returns의 λ 가 3, 2. 라고 하자. 각 location에서.

문제를 간단히 하기 위해, 각 location에서 20 cars 넘지 못한다고 하자. 추가적인 returned 차는 사라지는겨. 그리고 하룻밤에 옮길 수 있는 최대는 5대.
discount rate γ = 0.9 로 하고,
continuing finite MDP로 하자.
time steps는 days 고
state는 날의 마지막에 각 location에 있는 차량 숫자.
actions은 밤 사이 이동한 차량 net numbers.
(Fig4.2)는 차량을 하나도 옮기지 않는 policy에서 시작한 policy iteration 에 의한 policies의 sequence를 보여준다.

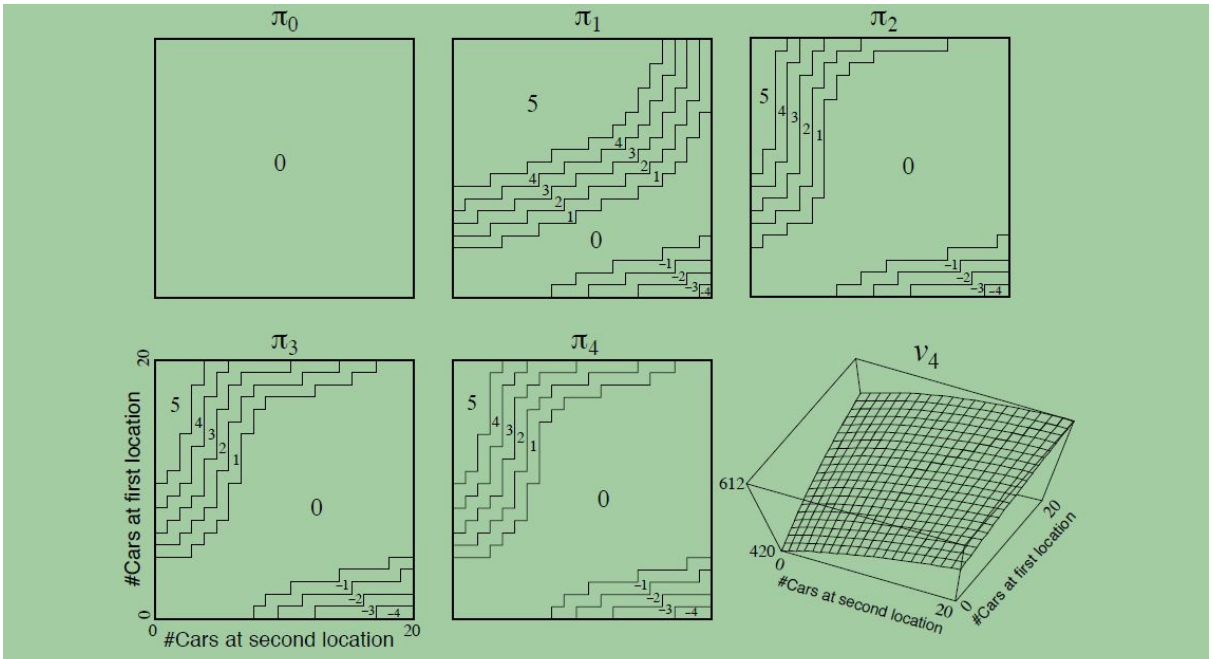


Figure 4.2: The sequence of policies found by policy iteration on Jack's car rental problem, and the final state-value function. The first five diagrams show, for each number of cars at each location at the end of the day, the number of cars to be moved from the first location to the second (negative numbers indicate transfers from the second location to the first). Each successive policy is a strict improvement over the previous policy, and the last policy is optimal.

(Fig4.2): policy iteration으로 찾은 policies의 sequence, state-value function.
첫 5개 diagram은 각 location에서 날의 마지막에 cars의 숫자에 대해, first location에서 다른 곳으로 옮겨질 차량의 숫자를 보여준다.
negative numbers는 second에서 first로.
각 연속적인 policy는 strict improvement를 보여준다.
마지막 policy가 optimal.
영역의 넓이는 뭐지?

(Exercise 4.5) (programming)
위 문제를 푸는 policy iteration 프로그램을 짜봐라. 살짝 문제를 바꿔볼게.
Jack의 직원 중 first location에 있는 애가 버스 타고 second location 근처에 산대.
그녀는 차 하나를 무료로 second location에 옮기겠대.
추가적인 차량은 여전히 2$ 의 cost가 들어.
그리고, Jack은 각 location에 parking space를 제한함.
그래서 만약 10대 이상이 밤새 남아있다면, 4달러의 추가비용 발생.
이런 비선형성과 arbitrary dynamics가 실제 문제에서는 종종 일어난다.

그리고 dynamic programming 이외의 optimization methods로는 쉽게 처리가 안 됨.

(Exercise 4.6) action values 에 대해 policy iteration을 정의해보렴.
q_* 를 구하는 complete algorithm을 내어 놓아라. v_* 하던 거랑 유사하게.
이건 특별히 중요하니까 꼭 해. 이 아이디어가 전체적으로 쓰임.

(Exercise 4.7) ε-soft policy 만 써야한다고 가정해보자.
그러니까, 각 s 에서 각 action을 선택할 확률이 최소한 ε / |A(s)| 이상이다.
v_* policy iteration algorithm과 다른 점을 steps 3, 2, 1 순서로 qualitatively 하게 나타내 보렴.

4.4 Value Iteration

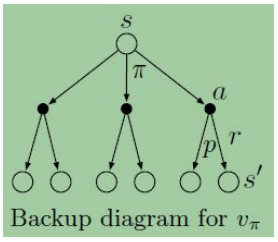
policy iteration의 drawback 하나는, 각 iteration이 policy evaluation을 포함하고 있다는 거다.
그러면 여러 state set 을 포함한 긴 반복 계산이 되겠지.
policy evaluation이 iteratively 하게 되면, v_π 로 정확하게 수렴하는 건 limit에서만 발생한다.
exact convergence를 위해 기다려야할까 아니면 그 전에 멈춰야할까?
(Fig 4.1)의 예는 policy evaluation을 자를 수 있다고 제안하고 있다.
그 예에서, 첫 세 번째 다음부터 policy evaluation iterations이 대응하는 greedy policy에 영향이 없다.

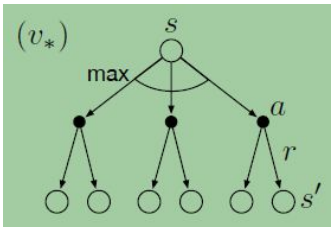
사실, policy iteration의 policy evaluation step은, policy iteration의 수렴성을 해치지 않고 여러 방식으로 잘릴 수 있다.
한 가지 중요한 special case는, policy evaluation이 one sweep(one update of each state) 이후 멈췄을 때다.
이 알고리즘을 value iteration이라고 한다.
value iteration 은 policy improvement와 truncated policy evaluation steps를 합친 간단한 update operation으로 나타낼 수 있다.
모든 s에 대해,

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_k(s')\Big], \end{aligned} \tag{4.10}$$

임의의 v_0에 대해, sequence {v_k}는 v_*로 수렴한다고 보일 수 있다. v_*의 존재성을 보장하는 조건과 동일한 조건 아래서.

value iteration을 이해하는 또 다른 방법은 Bellman optimality equation(4.1)을 참조하는 것.
value iteration 은 간단하게 얻어진다. Bellman optimality equation을 update rule로 바꾸는 거.
value iteration update 가 policy evaluation update(4.5)와 어떻게 identical 한지도 봐라.
모든 actions에 대해 maximum이 취해져야 한다는 것 빼고.
이 close relationship을 보는 또다른 방법은 backup diagrams을 보는 거다.





왼쪽이 policy evaluation, 오른쪽이 value iteration.
이 둘은 v_π, v_* 를 계산하는 natural backup operations 이다.

마지막으로, value iteration이 어떻게 terminate 되는지 보자.
policy evaluation과 마찬가지로, value iteration은 v_* 로 정확히 수렴시키기 위해 무한히 iteration 을 돌려야 한다.
In practice, value function이 한 sweep에서 조금만 바뀌면 stop 한다.
아래는 이런 종류의 termination condition이 반영된 complete algorithm이다.

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
  $\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$

Value iteration 이 매 sweep 마다 policy evaluation과 policy improvement 한 sweep씩 효율적으로 combine 하고 있다.
각 policy improvement sweep 사이에 여러 policy evaluation sweeps을 넣어 더 빠른 convergence가 가능하다.

일반적으로, truncated policy iteration algorithm의 전체 class는 sweeps의 sequence로 생각할 수 있다. 일부는 policy evaluation updates를 사용하고, 일부는 value iteration updates를 사용한다.
(4.10)의 max operation 이 이들 updates 간의 유일한 차이기 때문에, max operation이 policy evaluation의 일부 sweeps 에 add 됐다는 뜻일 뿐이다.
이 모든 알고리즘이 discounted finite MDPs 에서 optimal policy로 수렴한다.

(Example 4.3) Gambler's Problem
타짜가 coin flips 에 돈을 건다.
head면 그 flip에 걸린 돈만큼 딴다? tail 이면 잃고.
타짜가 100$ 따면 끝난다. 또는 돈을 다 잃으면 끝.
매 flip 마다 얼마나 걸지 정해야하는데..
이 문제는 undiscounted, episodic, finite MDP 라고 할 수 있지.
state는 타짜의 capital $s \in \{1, 2, ..., 99\}$
actions 은 stakes. $a \in \{0, 1, ..., \min(s, 100-s)\}$

reward는 goal에 도달했을 때 +1, 나머지 모든 transitions은 0.
state-value function은 각 state에서 이길 확률.
policy는 capital의 level에서 stakes로 가는 mapping.
optimal policy는 goal에 도달할 확률을 maximize 한다.
p_h 는 heads일 확률이라고 하자.
p_h 가 known 이면, 전체 문제가 known이고, 풀릴 수 있다. value iteration 같은 걸로.
(Fig 4.3)은 value function의 변화다. 연속적인 value iteration의 sweeps에 대해.
그리고 찾은 final policy도 보여준다. p_h = 0.4 일 때.
이 Policy는 optimal이지만 unique 하진 않다.
사실, optimal policies는 한 무데기가 있다. optimal value function에 대한 argmax action selection 이 동점인 것들?
그 전체 무더기 들이 어떻게 생겼나 짐작이 가누.

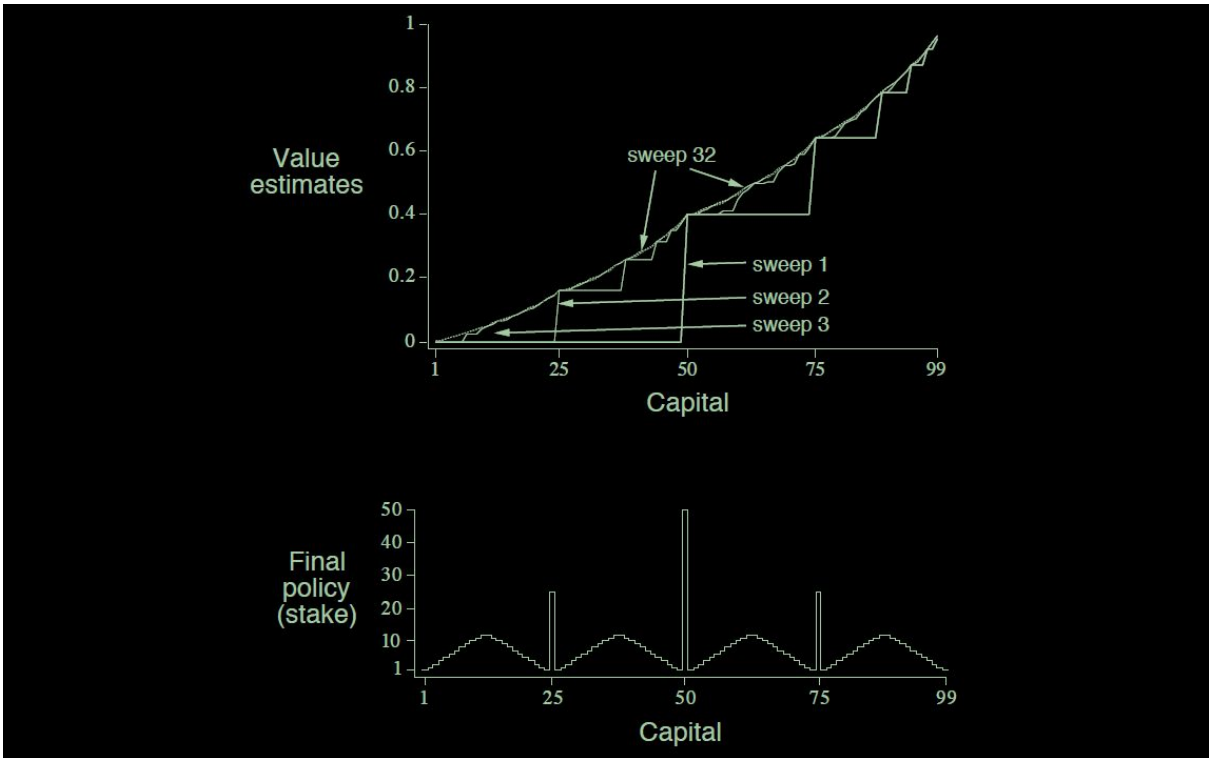


Figure 4.3: The solution to the gambler's problem for $p_h = 0.4$. The upper graph shows the value function found by successive sweeps of value iteration. The lower graph shows the final policy.

(Fig 4.3) p_h = 0.4 일 때의 solution.
위는 value iteration의 연속적인 sweeps으로 찾은 value function.
아래는 final policy.

(Exercise 4.8) 저 타짜 문제의 optimal policy는 왜 저렇게 희안하게 생겼을까?
특히, capital 이 50일 때 한 flip에 다 걸어. 근데 51에서는 아니지. 왜 이게 good policy일까?

(Exercise 4.9) (programming) p_h = 0.25, p_h = 0.55 일 때 value iteration 을 적용해봐라.
termination 에 해당하는 capital 0, 100에 0, 1을 주면서 2개의 dummy states를 쓰는 게 편리할 거야. 결과를 graph로 나타내보렴. θ 가 0으로 갈 때도 stable하니?

(Exercise 4.10) action values $q_{k+1}(s,a)$ 에 대한 value iteration update (4.10)의 analog는 뭐니?

4.5 Asynchronous Dynamic Programming
지금까지 논의한 DP methods의 주 결함은 MDP의 전체 state set에 대한 operations을 끼고 있다는 거지. 그러니까, state set 의 sweep이 필요하다는 것.
state set 이 매우 크면, sweep 한 번 하는 것도 피해야할 정도로 expensive할 수 있어.
예를 들어, backgammon 게임은 10^{20} states 가 있지.

Asynchronous DP algorithms 은 state set의 systematic sweeps 관점에서 in-place iterative DP algorithms 이다. 이런 알고리즘은 다른 states의 values 중 사용가능한 애들을 이용해 states의 value를 순서에 상관없이 states의 values를 update한다.
일부 states의 values는 여러 번 update 될 거다. 다른 values가 한 번 update 되기도 전에.
하지만 converge correctly 하기 위해선 asynchronous algorithm은 모든 states의 values를 update 해야한다.
계산상 특정 시점 이후로 어떤 state도 무시할 수 없다.
asynchronous DP algorithms은 update 할 states를 선택하는 데 엄청난 유연성을 준다.

예를 들어, asynchronous value iteration의 한 version은, updates the value, in place, of only one state, s_k , on each step, k, using the value iteration update (4.10).

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_k(s')\Big], \end{aligned} \tag{4.10}$$

만약 $0 \le \gamma < 1$ 이면, v_*로의 asymtotic(점근) convergence가 보장된다. 모든 states가 sequence $\{s_k\}$ 에서 infinite number of times 로 일어날 때만. (sequence는 stochastic이어도 가능)
(undiscounted episodic case 에서는, converge 하지 않는 updates의 ordering이 있을 수 있다. 하지만 비교적 피하기 쉽다.)
비슷하게, policy evaluation과 value iteration updates를 섞는 것도 가능하다. asynchronous truncated policy iteration의 한 종류를 produce 하기 위해.
이런 애들과 또 다른 unusual DP algorithms은 이 책의 범위를 벗어나지만,
다른 updates들이 다양한 sweepless DP algorithms에서 flexbly 쓸 수 있는 blocks 을 형성하는 건 clear 하지?

당연히, sweeps 을 피한다는 게 꼭 less computation을 의미하진 않는다.
단지 algorithm이 policy를 improving 하기 전에 hopeless한 long sweep 에 갇힐 필요가 없다는 걸 뜻할 뿐.

이 flexibility를 유용하게 쓸 수 있다. 알고리즘의 진행률을 향상시키기 위해 updates를 적용할 states를 선택함으로써.
우린 updates를 명령할 수 있다. value information이 state 간에 효율적으로 propagate되도록.
어떤 states는 다른 애들만큼 자주 그 values 가 updates 될 필요가 없을 수 있다..
일부 states는 updating을 아예 skip 할 수도 있겠지. optimal behavior와 관련이 없다면.
이걸 하는 아이디어는 Ch 8.에서 얘기하자.

Asynchronous algorithms은 computation을 real-time interaction과 intermix 하기도 쉽다.
주어진 MDP를 풀기 위해, agent가 실제 MDP 를 경험하는 것과 동시에 iterative DP algorithm을 돌릴 수 있다.
agent의 경험은 DP 알고리즘이 업데이트를 적용하는 states를 결정하는 데 쓰일 수 있다.
동시에, DP 알고리즘의 가장 최근 value와 policy information은 agent의 decision making을 가이드할 수 있다.
예를 들어, agent가 states를 방문할 때 states에 update를 적용할 수 있다.
이건 DP algorithms의 updates가, agent에 가장 relevant한 state set의 부분에 집중하게 해준다.
이런 종류의 focusing은 강화학습에서 반복되는 theme이다.

4.6 Generalized Policy Iteration

Policy iteration은 두 가지 simultaneous, interacting processes 로 이루어져 있다.

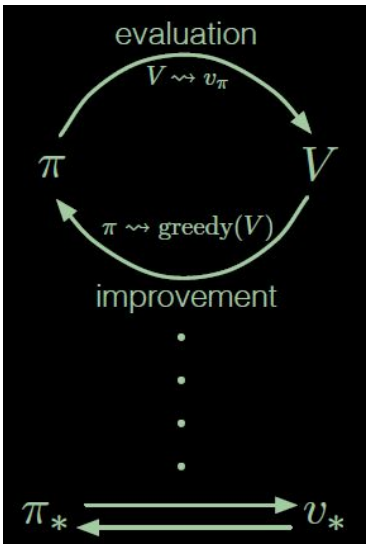
하나는 policy evaluation: value function이 current policy에 consistent 하게 만드는 것.
다른 하나는 policy improvement: current value function에 policy 를 greedy하게 만드는 것.

Policy iteration에선, 이 두 processes가 교대로 일어난다. 다른 하나가 시작하기 전에 하나가 completing한다.
하지만 이건 꼭 필요하진 않다.

Value iteration에선, 예를 들어, policy evaluation 한 번의 iteration이 각 policy improvement 사이에 수행된다.

asynchronous DP methods에선, evaluation과 improvement processes가 interleaved at an even finer grain. 뭔 소릴까.
어떤 case에서는 a single state 가 한 process에서 update 된다. 다른 애한테 return 되기 전에.
두 processes가 모든 states를 계속 update 하는 한, ultimate result는 보통 같다.
optimal value function과 optimal policy에 converge.

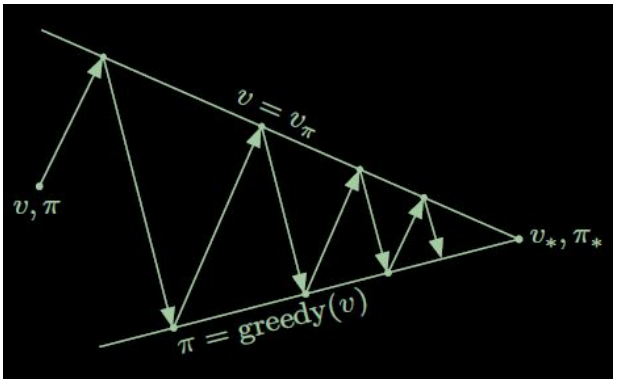
GPI: generalized policy iteration
이라는 용어를 쓴다. 어떨 때?
policy evaluation과 policy improvement 가 interact 하는 general idea를 말할 때.
두 processes의 granularity 나 다른 detail과는 독립적으로.
거의 대부분의 강화학습 methods 는 GPI로 잘 묘사된다.
그러니까, 모두 identifiable policies와 value functions를 갖고 있다는 말이고,
policy는 value function에 대해 improve 되고 있단 말이고,
value function은 항상 policy의 value function을 향한다?
그걸 묘사한 게 아래 그림.



두 processes가 stabilize하면, 그러니까, 더이상 changes 가 없으면,
value function과 policy 가 optimal 하다.
value function은 오직 current policy와 consistent 할 때 stabilize 한다.
policy는 오직 current value function에 greedy 할 때 stabilize 한다.
그래서, 두 processes 는 policy가 그 자체의 evaluation function에 대해 greedy 할 때 stabilize 한다.
이건 Bellman optimality equation(4.1) 을 만족한다는 걸 의미하고,
그러므로, policy 와 value function이 optimal 하다.

GPI 의 evaluation, improvement processes는 competing하고 cooperating 하는 두 가지 면을 갖고 있다.
opposing direction으로 끌어당기는 면에서 compete 하다.
policy를 value function에 대해 greedy 하게 만드는 건 보통 value function이 바뀐 policy에 대해 incorrect 하게 만들고, value function을 policy에 consistent하게 만드는 건 보통 policy가 더 이상 greedy 하지 않게 만든다.
하지만 long run에서, 두 processes는 single joint solution(optimal value function, policy)을 찾기 위해 interact 한다.

GPI 의 evaluation, improvement processes 간의 interaction을 두 constraints나 goals의 관점에서 생각할 수도 있다.
예를 들어, 2차원 공간에 두 직선이 있다고 하자. 아래 그림처럼.



실제 기하는 이것보다 더 복잡하지만, 실제 case에서 뭐가 일어나는지 잘 보여준다.

각 process는 value function과 policy를, 두 goals 중 하나의 solution을 나타내는 선 중 하나로 유도한다.
goals은 두 선이 orthogonal 하지 않기 때문에 interact 한다.
Driving directly toward one goal 하면 다른 goal에서 멀어진다.
하지만 inevitably, joint process는 전체적인 optimality goal에 가까워진다.
이 도표의 화살표는 각각 두 가지 목표 중 하나를 완전히 달성하는 데까지 시스템을 가져간다는 점에서 policy iteration의 behavior에 해당한다.
GPI에서 각 goal에 smaller, incomplete steps를 취할 수도 있다.
어느 case든, 두 processes는 optimality를 directly 달성하려고 하진 않지만 optimality의 overall goal을 함께 달성한다.

4.7 Efficiency of Dynamic Programming

DP 는 큰 문제에 적합하지 않을 수도 있다.
하지만 MDPs 를 푸는 다른 methods들과 비교했을 때, 확실히 효율적이다.
몇 가지 기술적 details을 무시하면,
DP methods가 optimal policy를 찾는 데 걸리는 시간은, states와 actions의 다항식이다. (최악의 경우에)
이 말은, n, k 가 states 와 actions의 수라고 하면, DP method가 n, k의 다항식보다 적은 연산을 수행한다는 뜻이다.
DP method는 polynomial time 내에 optimal policy를 찾는 것이 보장된다. 설령 (deterministic)policies의 수가 k^n 이라도.
이런 의미에서, DP는 policy space에서 어떤 direct search 보다 exponentially faster 하다.
왜냐하면, direct search는 같은 guarantee를 제공하기 위해 exhaustively examine해야할 것이기 때문이다.

Linear programming methods 또한 MDPs 를 풀기 위해 사용할 수 있다.
일부 case에선 이들의 worst-case convergence guarantees가 DP methods 보다 좋을 때도 있다.
하지만 linear programming methods는 훨씬 작은 수의 states에서 DP보다 비실용적으로 된다. (100 정도 쯤?)
큰 문제들에서 오직 DP methods만 feasible하다.

DP 는 가끔 applicability가 제한적이라고 여겨진다. curse of dimensionality 때문에.
state variables 의 수에 따라 states의 수가 exponentially 증가한다는 거지.
큰 state sets은 어려움을 만들어내지만, 이들은 문제의 어려움에 내재하는 것들이고, solution method로서의 DP의 어려움은 아니다.
사실, DP는 large state spaces를 다루는 데 비교적 좋다. direct search나 linear programming보다.

실제로는, 수백만 states 의 MDP를 풀기 위해 DP methods를 사용할 수 있다.
policy iteration, value iteration 모두 넓게 쓰이고, 일반적으로 뭐가 더 나은지는 불분명하다.
실제로는,이 methods들은 이론적 worst-case보다 빨리 수렴한다.
특히 good initial value functions이나 policies로 시작하면.

large state spaces 문제에서, asynchronous DP methods를 선호한다.
asynchronous method 를 one sweep 완료하는 데에도 모든 state에 대해 computation과 memory가 필요하다.
어떤 문제에서는 이런 많은 memory와 computation이 비실용적이지만, 잠재적으로 문제는 풀 수 있다. 왜냐면 optimal solution trajectories를 따라 발생하는 states는 상대적으로 적기 때문.
Asynchronous methods와 GPI 의 다른 variations가 이런 case에 적용될 수 있고 좋은, 또는 optimal policies를 synchronous methods 보다 빠르게 찾는다.

4.8 Summary

이 챕터에서는 dynamic programming의 기본 ideas와 algorithms를 알아봤다.
finite MDPs 를 푸는 데 관련이 있기 때문이지.
Policy evaluation 은 주어진 policy에서 value function의 iterative computation을 말한다.
Policy improvement는 해당 policy의 value function이 주어졌을 때 improved policy 의 computation을 말한다.
이 두 computations을 종합하면, policy iteration과 value iteration을 얻는다.
이 두 개가 DP methods 중에 가장 popular 하지.
이들 중 하나가 MDP에 대한 complete knowledge가 주어진 finite MDPs의 optimal policies와 value functions을 reliably 계산하는 데 쓰일 수 있다.

Classical DP methods 는 state set 전체를 sweep하는 operate를 하는데, *expected update* operation을 각 state에 대해 수행한다.
각 operation은 한 state의 value를 update한다. 가능한 모든 successor states와 그들의 발생확률에 기초해서.
Expected updates는 Bellman equations 과 밀접한 관련이 있다.
그들은 이 방정식들이 assignment statements(할당문?)으로 바뀐 것에 지나지 않는다.

updates가 더이상 value를 바꾸지 않을때, 대응하는 Bellman equation을 만족하는 value로 convergence가 발생.
4가지 주요 value functions(v_π, v_*, q_π, q_*) 가 있는 것처럼,
4가지 대응되는 Bellman equations,
4가지 대응되는 expected updates가 있다.
DP updates의 operation의 직관적인 관점은 backup diagrams 에서 주어진다.

DP methods, 아니, 모든 강화학습 methods들에 대한 insight는,
걔들을 generalized policy iteration(GPI) 로 봄으로써 얻을 수 있다.
GPI 는 approximate policy와 approximate value function을 중심으로 서로 상호작용하는 두 processes 의 general idea이다.
한 process는 policy를 주어진 것으로 보고 어떤 형태의 policy evaluation 수행한다. value function이 policy의 true value function처럼 되도록 바꾸면서.
다른 process는 value function이 given이고 policy improvement 형태를 수행하면서, policy를 더 좋게 바꾼다. value function이 true value function이라는 가정 하에.

각 process가 서로의 basis 를 바꾸지만, 전체적으로 얘들은 joint solution을 찾기 위해 함께 일해.

여기서 joint solution은 어느 한 process 에 의해 변하지 않는, 결론적으로 optimal인 policy와 value function 을 말해.
어떤 cases에선 GPI가 converge 한다고 증명될 수 있는데, 특히 이 챕터에서 다뤘던 classical DP 에선 더욱 그렇다.
다른 cases 에선 convergence가 증명되지 않았는데, 여전히 GPI 의 idea는 methods에 대한 이해를 향상시켜준다.

반드시 state set 을 통한 complete sweeps 에서 DP methods를 수행할 필요는 없다.
Asynchronous DP methods 가 임의의 순서로 states를 update하는 in-place iterative methods 들이고, 아마 확률적으로 determined 되며 out-of-date 정보를 이용한다.
이런 methods 중 많은 애들은 GPI의 fine-grained forms 으로 볼 수 있다.

마지막으로, DP methods의 last special property를 보자.
이들 모두는 states의 values의 estimates를 update 한다. successor states의 values의 estimates에 기초해서.
그 말은, 다른 estimates로 estimates를 update한다는 말.
이런 idea를 bootstrapping 이라 부른다.
많은 강화학습 methods는 bootstrapping을 수행한다. DP가 요구하는 것처럼 environment의 완전하고 정확한 모델을 필요로 하지 않는 방법이라도.
다음 챕터에서 model과 bootstrap 하지 않는 강화학습의 methods들을 살펴본다.
그 다음 챕터에선 model을 필요오 하진 않지만 bootstrap 하는 methods를 볼 거고.
이런 key features와 properties는 separable하다. 하지만 흥미로운 조합으로 섞을 수 있다.
아주 기대가 되는구만.