MDP에 대한 모든 정보를 알 때, 최적의 폴리시를 찾아나가는 방법.

Lecture 3: Planning by Dynamic Programming

방법론

David Silver

Outline

What is Dynamic Programming?

Dynamic sequential or temporal component to the problem

Programming optimising a "program", i.e. a policy

- c.f. linear programming

- A method for solving complex problems 복잡한 문제를 풀 방법론.
- By breaking them down into subproblems 큰문제를 작게 쪼갠다.
  - Solve the subproblems 쪼갠 것을 푼다.
  - Combine solutions to subproblems 조합한다.

Requirements for Dynamic Programming

* RL
Modelfree based
planning by D.P

Dynamic Programming is a very general solution method for problems which have two properties:

- Optimal substructure 최적의 해결책이 나눠진 작은 문제에 적용이 되어야 한다
  - *Principle of optimality* applies
  - Optimal solution can be decomposed into subproblems
- Overlapping subproblems
  - Subproblems recur many times 많이 등장하니까 한 subproblem 을 풀어서 저장해두면
  - Solutions can be cached and reused 다시 사용할 수 있다. (다시 만나기 때문)
- Markov decision processes satisfy both properties MDP가 위 두 property를 만족한다.
  - Bellman equation gives recursive decomposition
  - Value function stores and reuses solutions

Planning by Dynamic Programming

env 가 동작하는 것을 알고있음

- Dynamic programming assumes full knowledge of the MDP
- It is used for *planning* in an MDP
- For prediction: vf 을 찾음
  - Input: MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and policy π
    얘가 좋은지 나쁜지 optimal 인지 상관도 안함
  - or: MRP $\langle \mathcal{S}, \mathcal{P}^{\pi}, \mathcal{R}^{\pi}, \gamma \rangle$
  - Output: value function v_{π}
- Or for control: policy 를 찾음
  - Input: MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$
  - Output: optimal value function v_*
  - and: optimal policy π_*

Other Applications of Dynamic Programming

Dynamic programming is used to solve many other problems, e.g.

- Scheduling algorithms
- String algorithms (e.g. sequence alignment)
- Graph algorithms (e.g. shortest path algorithms)
- Graphical models (e.g. Viterbi algorithm)
- Bioinformatics (e.g. lattice models)

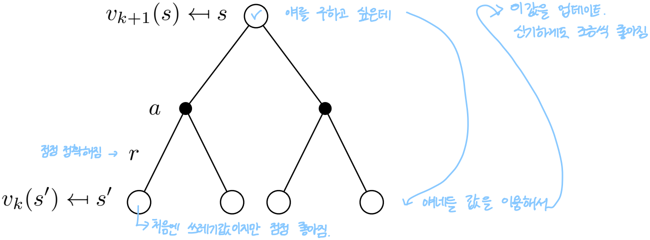
Iterative Policy Evaluation

폴리쉬를 평가하자~

- 이 policy를 따라갔을 때 return을 얼마 받냐 = policy Evaluation = value function
- prediction 문제이다

- Problem: evaluate a given policy π
- Solution: iterative application of Bellman expectation backup
  을 반복 실행 ↳ 잠깐 저장하는 것
- $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_\pi$ 궁극적 목표
- Using *synchronous* backups,
  한단계에 한 개씩
  - At each iteration $k + 1$
  - For all states $s \in \mathcal{S}$ 모든 state 를 한 번씩 업데이트
  - Update $v_{k+1}(s)$ from $v_k(s')$ 뒤에 값 $v_k(s')$ 을 이용해서 $v_{k+1}(s)$ 을 업데이트
  - where s' is a successor state of s
- We will discuss *asynchronous* backups later
- Convergence to v_π will be proven at the end of the lecture

Iterative Policy Evaluation (2)



Bellman expectation equation

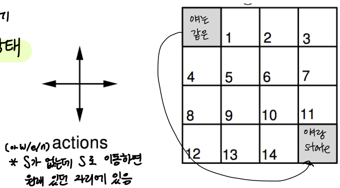
$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

$$\mathbf{v}^{k+1} = \mathcal{R}^{\boldsymbol{\pi}} + \gamma \mathcal{P}^{\boldsymbol{\pi}} \mathbf{v}^k$$

매 iterative 마다 위 식을 이용하여 모든 S 를 업데이트

Evaluating a Random Policy in the Small Gridworld

prediction 문제이기 때문에
policy 와 MDP 가 주어진 상태



Q. 1칸에서 14칸으로 갈 확률은? 1에서 3칸으로 갈 확률은?
A. 모른다. 운좋으면 ¼ⁿ 이겠지만, 엉뚱 돌수도 있으니까.

$r = -1$
on all transitions

↖ 이게 MDP 래

다시한번 문제 정리

MDP와 Policy 가 있는 상태. (prediction 문제)
value 를 찾는다
value 는 곧 평균 몇 칸을 지나야 종료할지이다.
(한번 움직일 때 마다 -1이 되니까)
value 는 return의 기댓값인데 return이 끝날 때 까지 총 리워드니까 그게 몇번 움직이는가가 된다.

- Undiscounted episodic MDP ($\gamma = 1$) S A P R γ 를 다 알고있으니까
- Nonterminal states $1, \ldots, 14$
- One terminal state (shown twice as shaded squares)
- Actions leading out of the grid leave state unchanged
- Reward is -1 until the terminal state is reached
- Agent follows uniform random policy 4가지 방향에 각 0.25 확률로 이동

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

a S

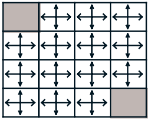
Iterative Policy Evaluation in Small Gridworld

일단 state 칸만큼 테이블을 만들어놓음

v_k for the Random Policy | Greedy Policy w.r.t. v_k

$k = 0$
(0으로 초기화)

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

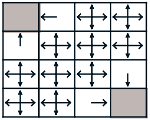


random policy

왜 이렇게 되느냐? Bellman equation 에 따라서!

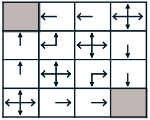
$k = 1$

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



$k = 2$

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



사실 우리는 굉장히 멍청한 policy를 평가한 것인데, 이 평가된 value 안에서 greedy 하게 움직이면 결국에 optimal policy가 찾아질 수 있다.

평가만 했는데! optimal policy 가 생겼다!

$$V^{k+1} = R^{\pi} + \gamma P^{\pi} V^{k}$$

γ: maybe 1

계산

내 피셜임

$$V^{1} = -1 + 1\left(\frac{1}{4}\cdot 0 + \frac{1}{4}\cdot 0 + \cdots + \frac{1}{4}\cdot 0\right)$$

$$V^{2} = -1 + 1\left(\frac{1}{4}\cdot(-1) + \cdots + \frac{1}{4}(-1)\right)$$

계산

$$V^{2} = -1 + \gamma\left(\frac{1}{4}(-1) + \frac{1}{4}(0) + \frac{1}{4}(-1) + \frac{1}{4}(-1)\right)$$

위로 가려다가 자신한테 가니까

$$= -1 - \frac{3}{4} = -1.75$$

반올림한거임

계산할게 맞는 식인지는 모르겠음.
괄호 안에가 이상한것 같아..

Iterative Policy Evaluation in Small Gridworld (2)

이 문제에서는 $k=3$ 까지만 해도 Optimal policy 가 나온다

$k = 3$

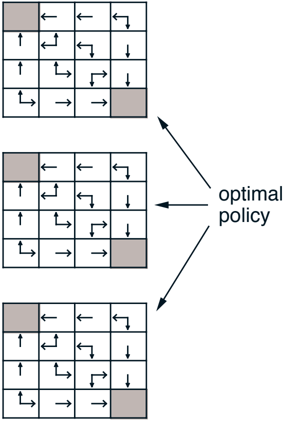
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

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

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0



Evaluate → Improve → Evaluate → Improve . . .

How to Improve a Policy

- Given a policy π
  - Evaluate the policy π value function 을 찾겠다.

$$v_\pi(s) = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + ... | S_t = s\right]$$

  - Improve the policy by acting greedily with respect to v_π

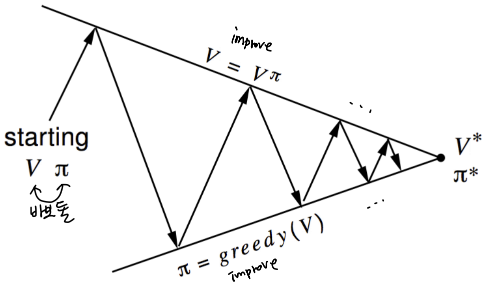
찾은 v_π에 대해 π'을 만든다.

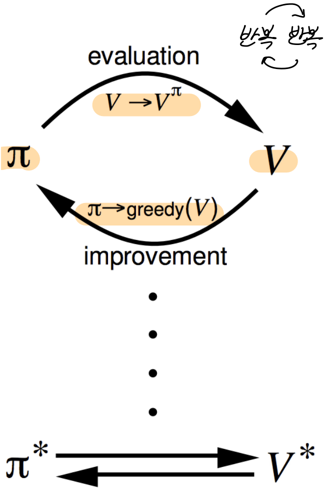
$$\pi' = \text{greedy}(v_\pi)$$

방금 예제는 너무 간단해서 π'가 바로 π^*이 됐다. 보통은 여러 번 해야 π^*에 도달

- In Small Gridworld improved policy was optimal, $\pi' = \pi^*$
- In general, need more iterations of improvement / evaluation
- But this process of policy iteration always converges to $\pi*$

Policy Iteration





Policy evaluation Estimate v_π

Iterative policy evaluation

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

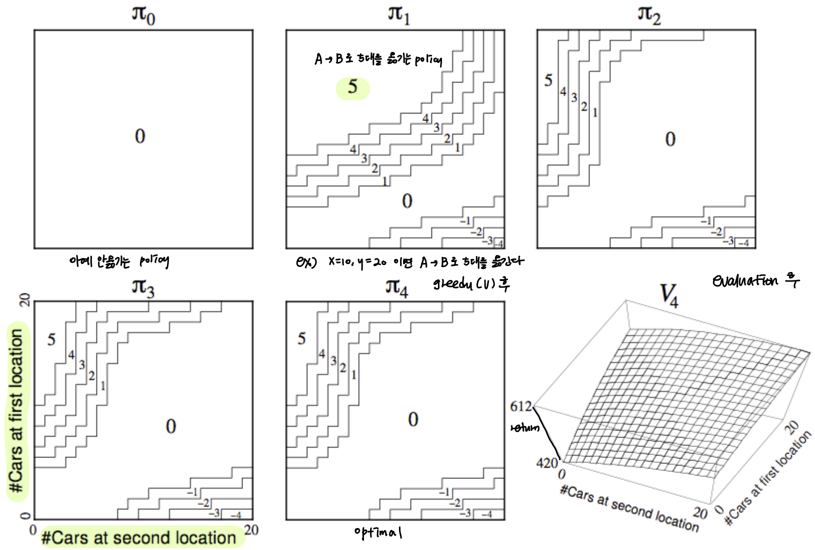
Greedy policy improvement

Jack's Car Rental

렌트해주는 loc 2개.

- States: Two locations, maximum of 20 cars at each
  두 개의 렌트장이 있고, 한 곳에는 최대 20개의 차가 존재할수 있다
- Actions: Move up to 5 cars between locations overnight
  A 와 B 간 차 이동이 가능하다.
- Reward: \$10 for each car rented (must be available)
- Transitions: Cars returned and requested randomly
  - Poisson distribution, n returns/requests with prob $\frac{\lambda^n}{n!}e^{-\lambda}$
    정해진 단위시간동안 사건이 발생할 확률
  - 1st location: average requests = 3, average returns = 3
    평균 하루에 3번 렌트 요청　평균 하루 3번 리턴
  - 2nd location: average requests = 4, average returns = 2

Policy Iteration in Jack's Car Rental



Policy Improvement

여기서 증명하고자 하는 것 : 정말 policy improvement 대로 하면 더 나은 policy 가 되는가?
결론 : 좋다!

- Consider a deterministic policy, $a = \pi(s)$ 어느 state 에 오면 정해진 action을 하는 policy
- We can *improve* the policy by acting greedily Q에 대해 greedy 하게 움직임

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\ q_\pi(s, a)$$

우변에서 뭘 하던,
같은 s에서의 max 값보다
작거나 같을 수 밖에

- This improves the value from any state s over one step,

greedy policy가 내가 할수 있는 것 중에
가장 좋은 것을 하는것이기에

action value function

(one step 에 대해) $$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s)$$

s에 있을 때 π 로 a를 고르고,
$q_\pi(s,a)$ 로 $v_\pi(s)$ 와 같다. (q의 정의 상 당연)

π를 따랐을 때 얻는 v

s로부터 π를 따라가나
s에서 π가 골라준 a을
하나 하고 그 뒤로 π를
따라가나는 같다

- It therefore improves the value function, $v_{\pi'}(s) \geq v_\pi(s)$

* 첫 스텝만 π'을 따르고 뒤는 π. by definition

$$\begin{aligned}
v_\pi(s) \leq q_\pi(s, \pi'(s)) &= \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s\right] \\
&\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s\right] \\
&\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, \pi'(S_{t+2})) \mid S_t = s\right] \\
&\leq \mathbb{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \ldots \mid S_t = s\right] = v_{\pi'}(s)
\end{aligned}$$

정의에 의해 우변과 좌변이 같다.

$v_\pi \rightarrow p_\pi$ 로 변환시키던 공식을 생각해보니.

아무튼,, 그렇다.

(그걸 반복적으로 적용해서 결국 더 좋아지는 결론.)

Policy Improvement (2) 계속 나아지면 수렴 포인트는 optimal 이다

- If improvements stop,

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s, a) = q_\pi(s, \pi(s)) = v_\pi(s)$$

- Then the Bellman optimality equation has been satisfied

논리 : 위 식이 만족되므로 optimal policy 다

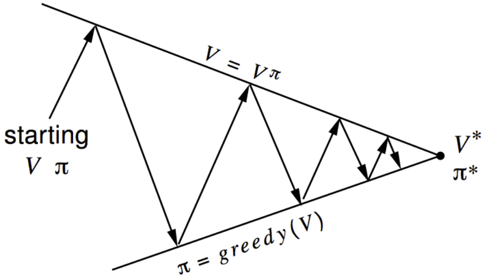
$$v_\pi(s) = \max_{a \in \mathcal{A}} q_\pi(s, a)$$

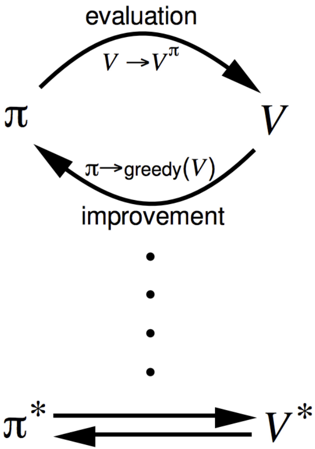
- Therefore $v_\pi(s) = v_*(s)$ for all $s \in \mathcal{S}$
- so π is an optimal policy

Modified Policy Iteration

- Does policy evaluation need to converge to v_π? v_π로 수렴할 때까지 해야되나
- Or should we introduce a stopping condition 스탑을 만들어야 하나
  - e.g. ϵ-convergence of value function OR
- Or simply stop after k iterations of iterative policy evaluation? k번만?
- For example, in the small gridworld $k = 3$ was sufficient to achieve optimal policy k번만 해도 충분하다.
- Why not update policy every iteration? i.e. stop after $k = 1$
  - This is equivalent to *value iteration* (next section) 해도 된다.

Generalised Policy Iteration





Policy evaluation Estimate v_π

Any policy evaluation algorithm

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

Any policy improvement algorithm

Principle of Optimality

땅요렙이 이해하지 못해 PASS!

Any optimal policy can be subdivided into two components:

- An optimal first action A_*
- Followed by an optimal policy from successor state S'

Theorem (Principle of Optimality)

A policy $\pi(a|s)$ achieves the optimal value from state s, $v_\pi(s) = v_*(s)$, if and only if

- *For any state s' reachable from s*
- *π achieves the optimal value from state s', $v_\pi(s') = v_*(s')$*

Deterministic Value Iteration

* 앞과 다른 점 : policy 가 없다.

- If we know the solution to subproblems $v_*(s')$ 이걸 알면
- Then solution $v_*(s)$ can be found by one-step lookahead 이걸 구할수있다

< Bellman Optimality Equation >

$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

- The idea of value iteration is to apply these updates iteratively
- Intuition: start with final rewards and work backwards 마지막에서부터계산.

  직관적
- Still works with loopy, stochastic MDPs → 결론적으로 optimal 에 수렴함.

  여기도 적용됨.

Example: Shortest Path

policy 가 없음!

주변 중에 max는 0.
따라서 변화없이 -1

g			

Problem

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

V_1

16개값을 다 업데이트 할건데 옛값의 식으로 계산.

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

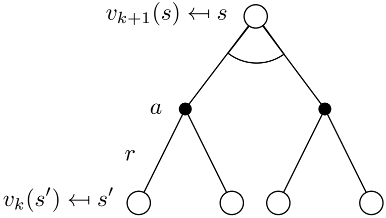
V_7

Value Iteration

- Problem: find optimal policy π
- Solution: iterative application of Bellman optimality backup
- $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_*$
- Using synchronous backups 각 반복마다 모든 s를 update
    - At each iteration $k+1$
    - For all states $s \in \mathcal{S}$
    - Update $v_{k+1}(s)$ from $v_k(s')$
- Convergence to v_* will be proven later
- Unlike policy iteration, there is no explicit policy
- Intermediate value functions may not correspond to any policy

중간중간에 나오는 value function은 어떠한 policy도 아니다.

Value Iteration (2)



$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

$$\mathbf{v}_{k+1} = \max_{a \in \mathcal{A}} \boldsymbol{\mathcal{R}}^{\mathbf{a}} + \gamma \boldsymbol{\mathcal{P}}^{\mathbf{a}} \mathbf{v}_k \quad \leftarrow \text{Bellman}$$

Example of Value Iteration in Practice

http://www.cs.ubc.ca/∼poole/demos/mdp/vi.html

한 time 에 모든 state update

Synchronous Dynamic Programming Algorithms

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 가 크다

- Complexity $O(mn^2)$ per iteration, for m actions and n states
- Could also apply to action-value function $q_\pi(s,a)$ or $q_*(s,a)$
- Complexity $O(m^2n^2)$ per iteration

Asynchronous Dynamic Programming

- DP methods described so far used *synchronous* backups
- i.e. all states are backed up in parallel 병렬
- *Asynchronous DP* backs up states individually, in any order
  모든 state가 아닌, 일부 / 개인만 backup.
- For each selected state, apply the appropriate backup
- Can significantly reduce computation
- Guaranteed to converge if all states continue to be selected
  이게 보장되어야 함

Asynchronous Dynamic Programming

Three simple ideas for asynchronous dynamic programming:

- *In-place* dynamic programming
- *Prioritised sweeping*
- *Real-time* dynamic programming

In-Place Dynamic Programming

코딩 테크닉에 가깝대요.

- Synchronous value iteration stores two copies of value function

테이블이 두개 필요.
old와 new

for all s in $\mathcal{S}$

$$v_{new}(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_{old}(s') \right)$$

$$v_{old} \leftarrow v_{new}$$

- In-place value iteration only stores one copy of value function

테이블을 하나만.
과거정보 필요가 없이

for all s in $\mathcal{S}$

$$v(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right)$$

Prioritised Sweeping

우선순위를 두어서 업데이트. 벨만 에러로 판별.
(가큰애로)

- Use magnitude of Bellman error to guide state selection, e.g.

$$\left| \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right) - v(s) \right|$$

- Backup the state with the largest remaining Bellman error
- Update Bellman error of affected states after each backup
- Requires knowledge of reverse dynamics (predecessor states)
- Can be implemented efficiently by maintaining a priority queue

Real-Time Dynamic Programming

s가 많지만 agent가 별개 안갈 때,
agent가 놀 때 자주가는 s만 먼저 업데이트

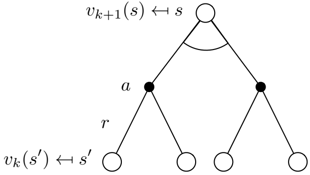
- Idea: only states that are relevant to agent
- Use agent's experience to guide the selection of states
- After each time-step S_t, A_t, R_{t+1}
- Backup the state S_t

$$v(S_t) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}^a_{S_t} + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{S_t s'} v(s') \right)$$

Full-Width Backups 여태까지 한 것.

- DP uses *full-width* backups
- For each backup (sync or async)
  - Every successor state and action is considered
  - Using knowledge of the MDP transitions and reward function

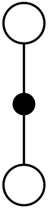
모든 갈 수 있는 s가 고려됨.
업데이트 시 참조하는 s들.



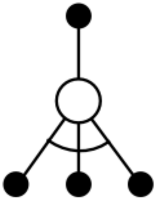
- DP is effective for medium-sized problems (millions of states)
- For large problems DP suffers Bellman's *curse of dimensionality* 큰문제에서 full-width는 X
  - Number of states $n = |\mathcal{S}|$ grows exponentially with number of state variables curse of dimensionality 문제
- Even one backup can be too expensive

Sample Backups

어디로 도착할지 몰라도, Backup이 가능한게 장점!



- In subsequent lectures we will consider *sample backups*
- Using sample rewards and sample transitions $\langle S, A, R, S' \rangle$
- Instead of reward function $\mathcal{R}$ and transition dynamics $\mathcal{P}$
- Advantages: ↗ 우리 여태 Model-based 였대. free는 a하고 어디갈지모름
  - Model-free: no advance knowledge of MDP required
  - Breaks the curse of dimensionality through sampling
  - Cost of backup is constant, independent of $n = |\mathcal{S}|$



silver

Approximate Dynamic Programming

강의에서 다루지 않습니다!

- Approximate the value function
- Using a *function approximator* $\hat{v}(s, \mathbf{w})$
- Apply dynamic programming to $\hat{v}(\cdot, \mathbf{w})$
- e.g. Fitted Value Iteration repeats at each iteration k,
    - Sample states $\tilde{\mathcal{S}} \subseteq \mathcal{S}$
    - For each state $s \in \tilde{\mathcal{S}}$, estimate target value using Bellman optimality equation,

$$\tilde{v}_k(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \hat{v}(s', \mathbf{w_k}) \right)$$

    - Train next value function $\hat{v}(\cdot, \mathbf{w_{k+1}})$ using targets $\{\langle s, \tilde{v}_k(s) \rangle\}$

Some Technical Questions

- How do we know that value iteration converges to v_*?
- Or that iterative policy evaluation converges to v_π?
- And therefore that policy iteration converges to v_*?
- Is the solution unique?
- How fast do these algorithms converge?
- These questions are resolved by *contraction mapping theorem*

Value Function Space

- Consider the vector space $\mathcal{V}$ over value functions
- There are $|\mathcal{S}|$ dimensions
- Each point in this space fully specifies a value function $v(s)$
- What does a Bellman backup do to points in this space?
- We will show that it brings value functions *closer*
- And therefore the backups must converge on a unique solution

Value Function ∞-Norm

- We will measure distance between state-value functions u and v by the ∞-norm
- i.e. the largest difference between state values,

$$||u - v||_\infty = \max_{s \in \mathcal{S}} |u(s) - v(s)|$$

Bellman Expectation Backup is a Contraction

- Define the *Bellman expectation backup operator* T^π,

$$T^\pi(v) = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v$$

- This operator is a γ-contraction, i.e. it makes value functions closer by at least γ,

$$\begin{aligned}
||T^\pi(u) - T^\pi(v)||_\infty &= ||\left(\mathcal{R}^\pi + \gamma \mathcal{P}^\pi u\right) - \left(\mathcal{R}^\pi + \gamma \mathcal{P}^\pi v\right)||_\infty \\
&= ||\gamma \mathcal{P}^\pi (u - v)||_\infty \\
&\leq ||\gamma \mathcal{P}^\pi ||u - v||_\infty||_\infty \\
&\leq \gamma ||u - v||_\infty
\end{aligned}$$

Contraction Mapping Theorem

Theorem (Contraction Mapping Theorem)

For any metric space $\mathcal{V}$ that is complete (i.e. closed) under an operator $T(v)$, where T is a γ-contraction,

- *T converges to a unique fixed point*
- *At a linear convergence rate of γ*

Convergence of Iter. Policy Evaluation and Policy Iteration

- The Bellman expectation operator T^π has a unique fixed point
- v_π is a fixed point of T^π (by Bellman expectation equation)
- By contraction mapping theorem
- Iterative policy evaluation converges on v_π
- Policy iteration converges on v_*

Bellman Optimality Backup is a Contraction

- Define the *Bellman optimality backup operator* T^*,

$$T^*(v) = \max_{a \in \mathcal{A}} \mathcal{R}^a + \gamma \mathcal{P}^a v$$

- This operator is a γ-contraction, i.e. it makes value functions closer by at least γ (similar to previous proof)

$$||T^*(u) - T^*(v)||_\infty \leq \gamma ||u - v||_\infty$$

Convergence of Value Iteration

- The Bellman optimality operator T^* has a unique fixed point
- v_* is a fixed point of T^* (by Bellman optimality equation)
- By contraction mapping theorem
- Value iteration converges on v_*