Chapter 3. Planning[1] by Dynamic Programming

1. Introduction

2. Prediction : Policy Evaluation

- 2-1. Iterative Policy Evaluation
- 2-2. Example : Small Gridworld

3. Control : Policy Iteration

- 3-1. Policy Iteration
- 3-2. Proof of Policy Improvement
- 3-3. Extensions to Policy Iteration

4. Control : Value Iteration

- 4-1. Value Iteration in MDPs
- 4-2. Example : Shortest Path
- 4-3. Summary of DP Algorithms

5. Extensions to Dynamic Programming

- 5-1. Asynchronous Dynamic Programming
- 5-2. Full-Width and Sample Backups

1) Model-Based인 상황에서 학습하는 것을 의미하며, Known MDP(=<Reward, State Transition Prob.>) 즉 Model을 input으로 받고 Policy를 Produce 또는 Improve하는 계산과정을 Planning이라 한다.

1. Introduction

#. 강화학습에서 Planning을 Dynamic Programming으로 해결하는 방법을 알아볼 것이다. 즉 Model을 Input으로 받고 Policy를 Producing 또는 Improve하는 과정을 Planning이라 하며, 구체적인 방법은 Prediction과 Control이 있는데 Prediction과 Control문제를 Dynamic Programming으로 푸는 방법을 배울 것이다.

a. Dynamic Programming은 큰 문제를 작은 문제로 나누고 이에 대한 Solution을 찾고 이를 통합하여 큰 문제를 푸는 방법론을 의미하며 이를 적용하기 위해서는 아래의 2가지 조건이 필요하다.

조건 1	**Optimal Substructure** • 전체 큰 문제의 Optimal Solution이 작은 문제로의 Sub-Optimal Solution으로 나뉠 수 있다.
조건 2	**Overlapping Subproblems** • 하나의 Sub-Problem을 풀면 이를 Cache(Back-up)에 저장하였다가 Reuse하거나 통합하여 전체 문제의 Optimal Solution을 구할 수 있다.

b. 이러한 개념을 강화학습과 연관 지어 보면, State (Action) Value가 Solution인 Bellman Equation을 재귀적으로 표현한 상태에서 중간 중간에 이 Values를 저장하고 Reuse한다는 의미로 해석할 수 있다.

c. Prediction과 Control의 개념은 아래와 같다.

		Input	Output
Planning	**Prediction**	$MDP\langle S, A, P, R, \gamma\rangle$ and π or $MRP\langle S, P^{\pi}, R^{\pi}, \gamma\rangle$	Value Function V_{π}
	Control	$MDP\langle S, A, P, R, \gamma\rangle$	Optimal Value Function V_{*} and Optimal policy π_{*}

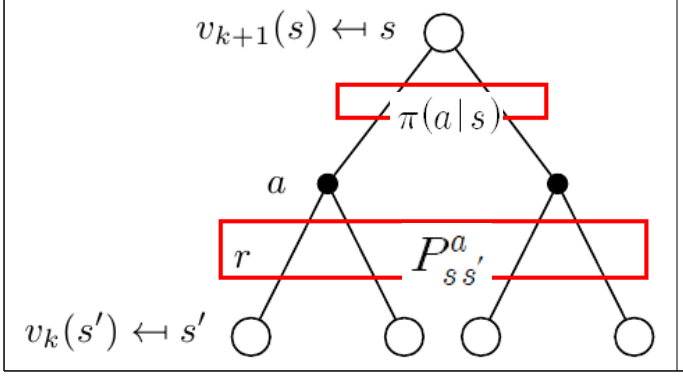
즉, Prediction은 MDP와 어떤 Policy 또는 MRP가 주어지고 그 Policy를 따랐을 때의 Value Function을 학습(계산)하는 것이며, Control은 MDP만 주어진 상황에서 Optimal Policy를 찾는 것이다.
Prediction의 경우 Update된 State Value Function을 바탕으로 Greedy하게 움직이면 Optimal Policy가 도출된다.

2. Prediction : Policy Evaluation

2-1. Iterative Policy Evaluation

#. Policy Evaluation은 한 State에서 주어진 Policy를 따라갔을 때, Expected Return, 즉 Value Function[2] 값을 구한다는 것이다. DP의 관점에서는 아래의 Bellman Expectation Equation을 이용하여 Synchronous Backup[3]을 반복적으로 하여 모든 State의 Value값이 수렴(v_*)할 때까지 Update[4]하는 방식이다. 이 때 Policy는 예를 들어 Random Policy로 주어졌다면, 계속 유지된다.

Bellman Expectation Equation



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

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

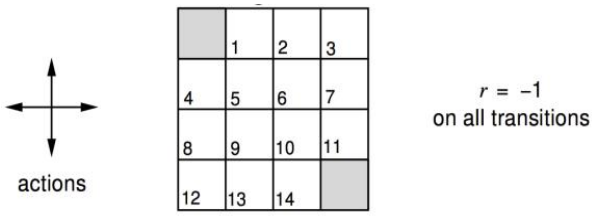
a. Bellman Expectation Equation은 한 Step 후에서 가능한 모든 State Value Functions의 평균값으로 현재 State Function을 Update하는 것이 핵심이다.

b. 초기 State Value는 쓰레기 값이지만, Recursive한 Bellman Expectation Equation을 이용하여 Terminal State의 부근부터 조금씩 정확한 값들이 확산되어가는 과정이다.

c. 1번의 Iteration에서 모든 State의 Value들을 Update하고 있으므로 Full-Swap이라고 하고 한 단계에서 모든 State하나씩 Update하므로 Synchronous Backup이라고 한다.

2-2. Example : Small Gridworld

a. 위에서 설명했던 내용을 Small GridWorld에 적용해보자. Discount factor $\gamma = 1$, Uniform Random Policy이다. $(\pi(n|\cdot) = \pi(e|\cdot) = \pi(s|\cdot) = \pi(w|\cdot) = 0.25)$



2) $v_\pi(s) = E[R_{t+1} + \gamma R_{t+2} + \ldots | S_t = s]$
3) Update시 모든 State의 Value를 Update하는 작업
4) 값을 조금씩 정확하게 만드는 작업

b. Bellman Expectation Equation과 Iteration을 이용한 계산 또한 아래와 같다.

$$-1 = \frac{1}{4}\times(-1+0) + \frac{1}{4}\times(-1+0) + \frac{1}{4}\times(-1+0) + \frac{1}{4}\times(-1+0)$$

$$-1.7 = \frac{1}{4}\times(-1+0) + 3\times\frac{1}{4}\times(-1+-1)$$

Iteration	v_k for Random Uniform Policy	Iteration	v_k for Random Uniform Policy
$k=0$	0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0	$k=3$	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
$k=1$	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	$k=10$	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
$k=2$	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	$k=\infty$	0.0 -14. -20. -22. -14. -18. -20. -20. -20. -20. -18. -14. -22. -20. -14. 0.0

c. 위와 같이 Evaluation된 State Value Function에서 Greedy하게 움직이면 Optimal Policy가 도출된다. 하지만 이는 간단한 문제라서 가능한 것임을 잊지 말자.

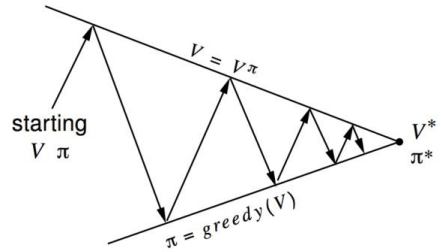
d. 위의 Example에선 Terminal State가 0.0으로 표시된 곳이며 Initial State는 Random하게 설정된다.

e. 구체적으로 구현하려면 env.step()을 통해 next_state를 구하고 이를 이용하여 State-Value를 구하면 된다. 즉 now_state = next_state로 인해 Agent의 Action이 적용되는 것이므로 env.step()을 통해 lookahead할 수 있다.

3. Control : Policy Iteration

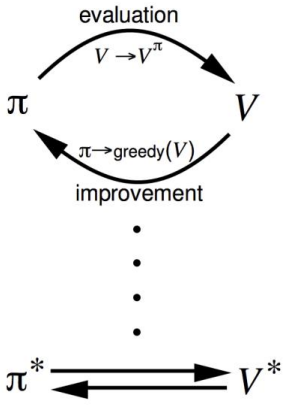
3-1. Policy Iteration

#. 처음에는 임의의 Policy가 있었는데, 이를 가지고 한 번 Policy Evaluation[5](일정 값에 수렴할 때 까지)하고 도출된 State Value에 대해 Greedy하게 움직이는 Policy를 새롭게 도출하고, 도출된 Policy를 가지고 다시 한 번 Evaluation하는 방법을 여러 번 거치면 Optimal Policy에 수렴할 수 있다.



Policy evaluation Estimate v_π
Iterative policy evaluation

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



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

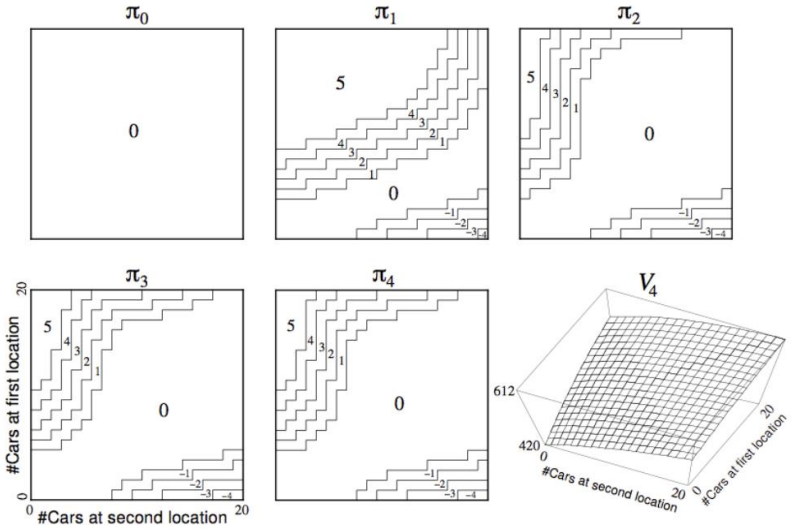
(위의 Pseudo Code에서 θ는 State Value Function이 수렴하는지 판단하기 위한 상수이다.)

5) 주어진 π을 Parameter로 한 State Value Function을 구한다는 의미 ($E_\pi[G_t \mid S_t = s]$)

a. Jack's Car rental Example[6]을 통해 Policy iteration을 알아보자.

States	한 렌트카 회사의 A, B 지점에서의 현재 보유하고 있는 차량의 수이며 각 지점은 최대 20대의 차를 보유할 수 있다.
Actions	각 지점 간 하루에 최대 5대의 차를 옮길 수 있다.
Reward	차 1대당 렌트가 발생하면 $10 이익이 난다.
Transitions	차의 반납과 렌트는 각 지점은 다른 푸아송 분포를 따른다. (외부요인)
	Poisson distribution, n returns/requests with prob $\frac{\lambda^n}{n!}e^{-\lambda}$ 1st location: average requests = 3, average returns = 3 2nd location: average requests = 4, average returns = 2

b. 문제의 상황은 위와 같으며, 목적은 회사의 이익을 극대화하기 위해 각 지점에 차량을 몇 대씩 옮겨야 하는지를 구하는, 즉 Optimal Policy를 찾는 것이다. Policy Iteration 방법을 적용한 결과는 아래와 같다.



c. 위의 그래프에서 각 State에 대한 Policy를 등고선으로 표현하였다. 즉 5인 곳에서는 A→B로 옮기는 Policy인 것이며 나와 있진 않지만, Value Function에 대해 Greedy하게 움직인 것을 등고선으로 표현한 것이다. 그리고 마지막 그래프에서 z축은 Value Function을 의미한다.

6) 디테일한 부분은 text book을 참고하면 된다.

3-2. Policy Improvement의 증명[7]

#. Policy Improvement에서 Policy Evaluation에서 구한 State Value에 대해 Greedy하게 하면 무조건 이전 Policy보다 좋은가를 증명하는 과정이다. 이를 위해 먼저 한 State에 있으면 특정 Action을 무조건 하는 Deterministic 상황[8]이라 가정하고, 한 Step에서 Greedy한 Action Value가 그렇지 않은 경우보다 더 높음을 보이고, 이후 모든 Step에서 Greedy한 Action State Value가 더 높음을 보이는 증명 방식이다.

a. 먼저 Greedy Policy를 아래와 같이 정의한다.

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

b. 그리고 첫 Step에선 $\pi'(s)$을 따라 Greedy하게 움직이고 이후에는 π을 따라 움직이는 Action Value와 처음부터 π을 따라 움직이는 Action Value를 비교하면 아래와 같다.

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

c. 이를 Iteration에 적용하여 모든 Step에 대해서 $v_{\pi'}(s) \geq v_\pi(s)$임을 아래와 같이 보이면 증명이 완성된다.

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

d. 결국 Greedy하게 Policy Improvement를 하면 된다는 것을 알았다. 하지만 언제까지 Improve를 해야 하는가? 이에 대한 답은 Optimal Value Function이라는 Upper Bound 때문에 더 이상의 Improvement가 되지 않을 때 까지 진행하면 된다. 이 상황에서는 Bellman Optimality Equation이 성립하는 상황이므로 $v_\pi(s) = v_*(s) \text{ for } all\, s \in S$이고 π가 Optimal Policy라 할 수 있다.

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

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

7) Greedy Algorithm의 한계와 비슷하지만 다른 점은 1. 일반적인 DP에서는 Value가 미리 주어져 있으며, 2. 동일한 Reward Scalar로 Value가 정해지기 때문.
8) State Value와 Action Value가 동일하다. ($q_\pi(s, \pi(s)) = v_\pi(s)$)

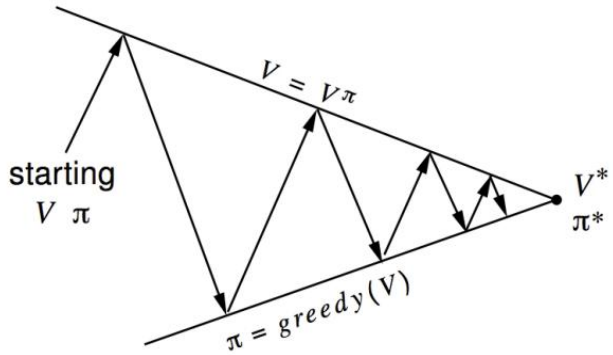
3-3. Extensions to Policy Iteration

3-3-1. Modified Policy Iteration

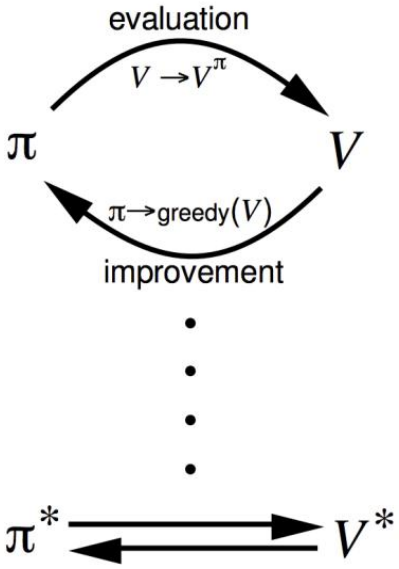
a. 2-2 Example에서 보았던 것처럼, Value Function이 Optimal하지 않은 상태에서도 이미 Optimal Policy는 구한 상태였다.($k=3$) 이처럼 굳이 Policy iteration에서 Policy Evaluation을 계속할 필요가 있는지에 대한 의문이 생긴다. 아니면 극단적으로 $k=1$만 Policy Evaluation을 하고 나머지에 대해선 계속 Greedy한 Policy Improvement를 하면 어떤가가 바로 Value Iteration의 아이디어이다.

3-3-2. Generalized Policy Iteration (GPI) in DP

a. DP의 관점에서 Policy Iteration을 일반화 하면, Optimal Value Function과 Optimal Policy를 구하기 위한 Frame-Work로 볼 수 있다. 즉 Policy Evaluation과 Policy Improvement 각각의 단계에서 알맞은 알고리즘을 사용하면 된다.



Policy evaluation Estimate v_π
Any policy evaluation algorithm
Policy improvement Generate $\pi' \geq \pi$
Any policy improvement algorithm

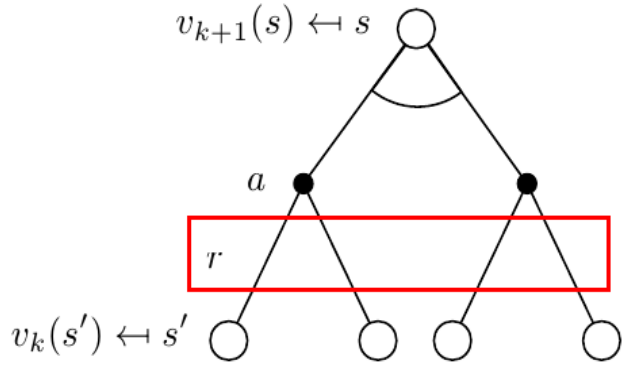


4-1. Value Iteration in MDPs

#. 2에서의 Policy Evaluation은 Bellman Expectation Equation을 사용하여 Optimal Policy를 구하는 방법이었다. Value Iteration에서는 Iteration과정 중 Policy Improvement 단계 없이 Bellman Optimality Equation을 사용하여 Policy없이[9] State Value Function만 수렴할 때까지 Update를 하고, 이를 이용하여 Optimal Policy를 구하는 방법론이며, 이는 Principle of Optimality에 근거를 두고 있다.

2에서와 동일하게 아래의 Bellman Optimality Equation을 이용하여 Synchronous Backup을 반복적으로 하여 모든 State의 Value값이 수렴(v_*)할 때까지 Update하는 방식이다.

Bellman Optimality Equation



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

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

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

a. Bellman Optimality Equation은 한 Step 다음에서의 가능한 Value Function 중 가장 큰 값으로 현재 값을 Update하는 것이 핵심이다. 또한 $\max$연산이 있기 때문에 역행렬로 계산이 불가능하여, Iterative 방법으로 풀어야 하는데, 이를 위해서는 Principle of Optimality 이론을 이용해야 한다.

b. Principle of Optimality는 Optimal State로 부터의 전파성이 핵심이며, 정리하면, s'에서의 Optimal Value를 알고 있다면, s'로 도달 가능한 s에서의 Optimal Value 또한 구할 수 있다는 것이다. 따라서 Iterative하게 계속 적용하면 s는 누군가의 s'이므로 계속해서 Optimal Value를 구할 수 있다.

9) Bellman Optimality Equation에는 원래 Policy $\pi(a|s)$가 없으며, $\max$연산이 등장한다.

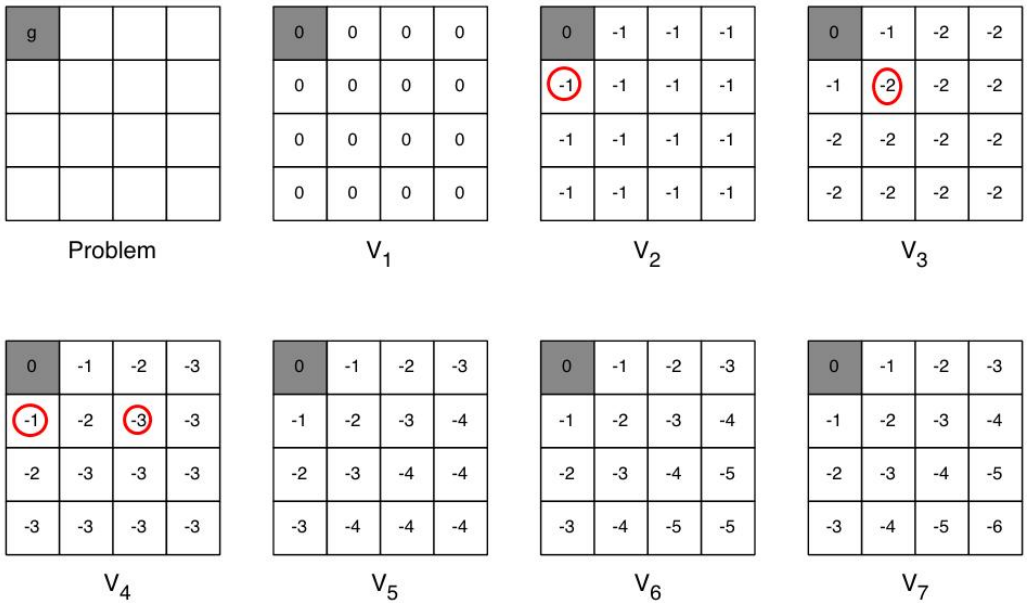
Theorem : Principle of Optimality
A policy $\pi(a\|s)$ achieves the optimal value from state s, $v_\pi(s) = v_*(s)$, if and only if • For any state s' reachable from s • π achieves the optimal value from state s', $v_\pi(s') = v_*(s')$

c. 직관적으로 생각하면, Terminal State 바로 직전에는 Optimal Value를 알기 때문에 이런 식으로 뒤로 전파해가면 되는 것이다. 결국은 DP와 같이 Sub-Problem의 Solution(=State Value Function)들을 Caching (back-up)해서 풀 수 있는 것이다.

d. 결국 Sub-Problem인 $v(s')$에서의 $v_*(s)$을 안다면, 한 Step-Look Ahead(Bellman Optimality Equation)를 통해서 $v_*(s)$을 구할 수 있다는 것이다.

4-2. Example : Shortest Path

a. 위에서 설명했던 내용을 Shortest Path에 적용해보자. 즉 Bellman Optimality Equation을 iterative하게 적용하여 풀이하는 것이다.



$$-1 = \max[(-1+1\times1\times0),(-1+1\times1\times0),(-1+1\times1\times0)]$$

$$-2 = \max[(-1+1\times1\times-1),(-1+1\times1\times-1),(-1+1\times1\times-1),(-1+1\times1\times-1)]$$

$$-3 = \max[(-1+1\times1\times-2),(-1+1\times1\times-2),(-1+1\times1\times-2),(-1+1\times1\times-2)]$$

$$-1 = \max[(-1+1\times1\times0),(-1+1\times1\times-2),(-1+1\times1\times-2)]$$

b. 위의 Iteration 과정을 통해 알 수 있듯이 Terminal State에 가까운 State부터 Update가 이루어진다. 이는 MDP가 주어졌기 때문에 가능한 일이라고 할 수 있으며, 다른 한편으로는 Terminal State의 존재 여부에 따라 Value Iteration의 사용 가능 여부가 달라짐을 알 수 있다.

c. 이러한 식으로 Update된 최종적인 Value Function을 통해서 Optimal Policy는 한 번의 Greedy를 통해 자연스레 도출된다.

d. Policy Evaluation과의 차이점은 State Value Update과정에서 Policy Evaluation에선 가능한 모든 다음 State에서의 결과에 대해 Expectation 연산을, Value Iteration에서는 가능한 모든 다음 State에서의 결과 중 Max연산을 사용하는 것이다.

4-3. Summary of DP Algorithms

a. Policy Iteration과 Value Iteration의 차이는 아래와 같다.

Policy Iteration	$v(s) = \sum_i \pi(a_i \mid s)(R_s^{a_i} + \gamma v(s'))$
Value Iteration	$v(s) = \max_{a \in A}(R_s^{a_i} + \gamma v(s'))$

Policy Iteration의 경우 Policy Improvement로 인해 $\pi(a|s)$가 Update되지만 Value Iteration의 경우 Update과정에서 Policy가 존재하지 않기 때문에 한 State s에서 가능한 모든 Action에 대해 각각을 선택하였을 경우 중 TD-Target값이 가장 큰 값으로 s을 Update한다.

b. 이번 단원에서는 하나의 Iteration에서 모든 State를 Update하는 Synchronous Back-up을 다루었으며 그 요약은 아래와 같다.

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

c. 알고리즘에서 다루는 Value Function의 종류에 따라 시간 복잡도는 아래와 같이 달라진다.

Algorithms based on	Time Complexity
State-Value Function $v_\pi(s)$ or $v_*(s)$	$O(mn^2)$
Action-Value Function $q_\pi(s,a)$ or $q_*(s,a)$	$O(m^2n^2)$

5-1. Asynchronous Dynamic Programming

#. DP문제를 풀 때, Practical하게 널리 쓰이는 방법이다. Asynchronous인 경우에는 아래와 같은 3가지 방법 중 하나를 택하여 사용하면 시간 복잡도를 굉장히 줄일 수 있다. 수렴함은 증명되어 있지만 이것이 보장되려면 여러 State들이 골고루 Sampling이 되어야 한다.

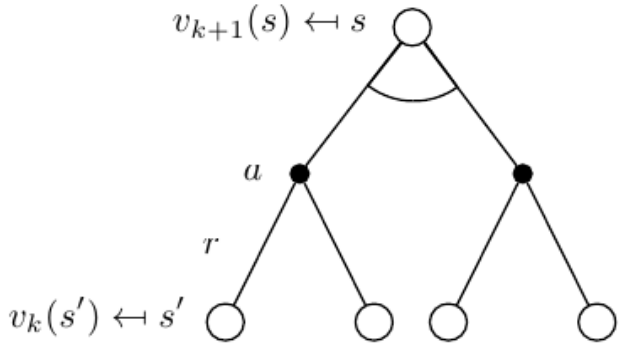
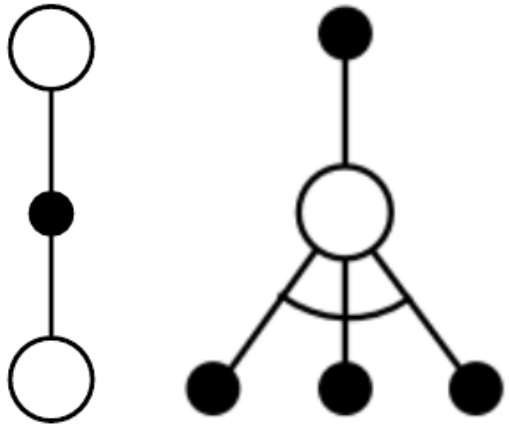
In-place DP	• 코딩 테크닉에 가깝다. Policy Iteration, Value Iteration을 구현하려면, Lookup Table이 old, new 2개가 필요했지만 하나의 Lookup Table이면 충분히 Update할 수 있는 방법론이다. $$v_{new}(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_{old}(s') \right)$$ $$v(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right)$$
Prioritised Sweeping	• Update할 때, 순서는 상관없기 때문에 Bellman Error[10]을 기준으로 이 값이 큰 애들이 중요한 애들이기 때문에 (S, A)을 우선수위 큐를 이용하여 아래의 Bellman Error(p) 값이 일정기준 이상이면 Priority p로 우선순위 큐에 넣고 먼저 Update하는 방법론이다. $$\left| \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right) - v(s) \right|$$
Real-Time DP	• Agent를 설정하여 Agent가 먼저 방문한 State들을 먼저 Update하는 방법론이다. Agent와 관련된 정보들을 S_t, A_t, R_{t+1}이라면 아래와 같이 Update한다. $$v(S_t) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_{S_t}^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{S_t s'}^a v(s') \right)$$

10) 이 값을 줄이는 방향으로 학습한다.

5-2. Full-Width and Sample Backups

#. Full-Width Backup의 경우 각 한번의 Backup 연산을 할 때, Sync, Async이든 s에서 갈 수 있는 모든 s'에 대해 Update를 하기 때문에 현실적으로 State가 많은 상황에서는 사용하기 어렵다. (차원의 저주) 따라서 Sample Backup을 사용하는데 이 방법은 아래와 같은 2가지 장점이 존재한다.

장점 1	고정된 비용으로 Backup이 가능하여 차원의 저주 문제가 발생하지 않는다.
장점 2	현실세계에서 더 많은 Model-Free[11]에 대해서도 적용이 가능하다. 예를 들어 특정 State에서 100개의 Action을 Sampling을 하여 Backup을 하는데, 100번의 Action으로 인해 도달한 State 중에는 중복 또한 존재할 수 있다. 결국 도달한 State를 이용하여 Backup을 진행하는 것이므로 Model-Free에서도 충분히 적용할 수 있다.

Full-Width Backup	**Sample Backup**
	

11) State에서 Action을 해도 어느 State로 이동할지 모르는 상황이다. 즉 Reward, State Transition Probability를 모른채로 학습하는 방법론이다.