Dynamic Programming

Dynamic Programming

- Dynamic programming is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems
- The optimal solution to the overall problem depends upon the optimal solution to its subproblems.

- Dynamic programming amounts to 1) breaking down a problem into simpler sub-problems, and 2) storing the solution to each sub-problem so that each sub-problem is reused in similar problems.

- Example:
  - $c(n,k) = \frac{n!}{k!(n-k)!}$
  - $c(n,k) = \begin{cases} c(n-1,k-1) + c(n-1,k), & \text{if } 0 < k < n \\ 1, & \text{if } k = 0 \text{ or } k = n \end{cases}$

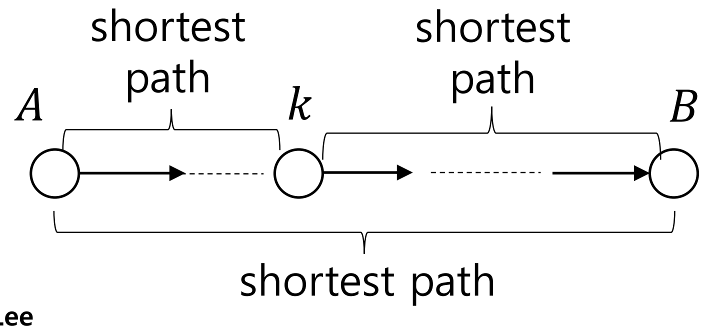
	0	1	2	3	4	k
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
n						

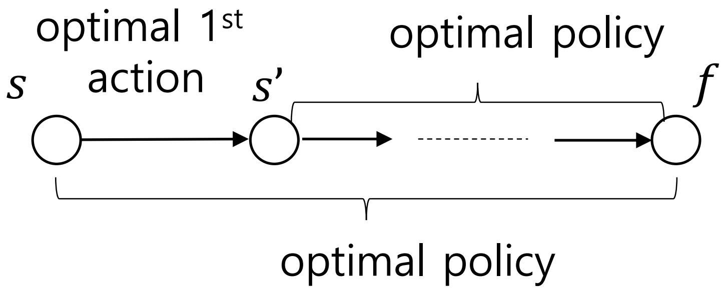
- Dynamic Programming is a lot like divide and conquer approach
- The difference is results of a sub-problem are used in similar sub-problems.

Principle of Optimality

- Dynamic Programming is a very general solution method for problems which have two properties:
  - Optimal substructure
    - Principle of optimality applies
    - Optimal solution can be decomposed into subproblems
  - Overlapping subproblems
    - Subproblems recur many times
    - Solutions can be cached and reused

- Markov decision processes satisfy both properties
  - Bellman equation gives recursive decomposition
  - Value function stores and reuses solutions





Prediction & Control in Dynamic Programming

- Planning in an MDP
- For prediction: Prediction is to find the value function by evaluating a policy using the Bellman Expectation Equation.
  - Input: MDP $< S, A, P, R, \gamma >$ and policy π
  - Output: value function v_π
- For control: This process involves optimizing the value function, we calculated during the prediction process.
  - Input: MDP $< S, A, P, R, \gamma >$
  - Output: optimal value function v_* and optimal policy π_*
- Dynamic programming assumes full knowledge of the MDP
  - Assume *transition prob* and *rewards* are known.
  - Feasibility in real-world engineering applications is therefore limited

Policy Evaluation (Prediction)

- Prediction problem: given a policy π, compute the state-values(v) of the model using the policy

Method 1) Using matrix $\boldsymbol{v}_\pi = \boldsymbol{R}_\pi + \gamma \boldsymbol{P}_\pi \boldsymbol{v}_\pi \Rightarrow \boldsymbol{v}_{\boldsymbol{\pi}} = (\boldsymbol{I} - \gamma \boldsymbol{P}_\pi)^{-\mathbf{1}} \boldsymbol{R}_\pi$
(from MDP chapter)

Method 2) Iterative approach: Iterative application of Bellman Expectation Equation

$$v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_\pi$$

Policy Evaluation (Prediction)

Matrix Method:

- Use Bellman expectation equation
- Recap: Bellman Expectation Equation:

$$v_\pi(s) = \sum_{a\in A} \pi(a|s)\left(R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a v_\pi(s')\right) = \underbrace{\sum_{a\in A} \pi(a|s)\, R_s^a}_{R_s^\pi} + \gamma \sum_{s'\in S} \underbrace{\sum_{a\in A} \pi(a|s) P_{ss'}^a}_{P_{ss'}^\pi} v_\pi(s')$$

- For every state s, define $v_\pi(s)$

$$\begin{bmatrix} v_\pi(1) \\ \vdots \\ v_\pi(n) \end{bmatrix} = \begin{bmatrix} R_1^\pi \\ \vdots \\ R_n^\pi \end{bmatrix} + \gamma \begin{bmatrix} P_{11}^\pi & \cdots & P_{1n}^\pi \\ \vdots & \vdots & \vdots \\ P_{n1}^\pi & \cdots & P_{nn}^\pi \end{bmatrix} \begin{bmatrix} v_\pi(1) \\ \vdots \\ v_\pi(n) \end{bmatrix}$$

$$\boldsymbol{v_\pi = R_\pi + \gamma P_\pi v_\pi \Rightarrow v_\pi = (I - \gamma P_\pi)^{-1} R_\pi}$$

Policy Evaluation (Prediction)

Iterative Method:

- When number of states is too big or continuous, matrix method is too complex or impossible
- During one iteration, the *old* value of s is replaced with a *new* value from the old values of the successor state s'
  - Update $v_{k+1}(s)$ from $v_k(s)$
- Algorithm
  - At each iteration *k+1*
  - For all states *s* ∈ *S*
    - Update v_{K+1}(s) from $v_k(s')$ where s' is a successor state of s

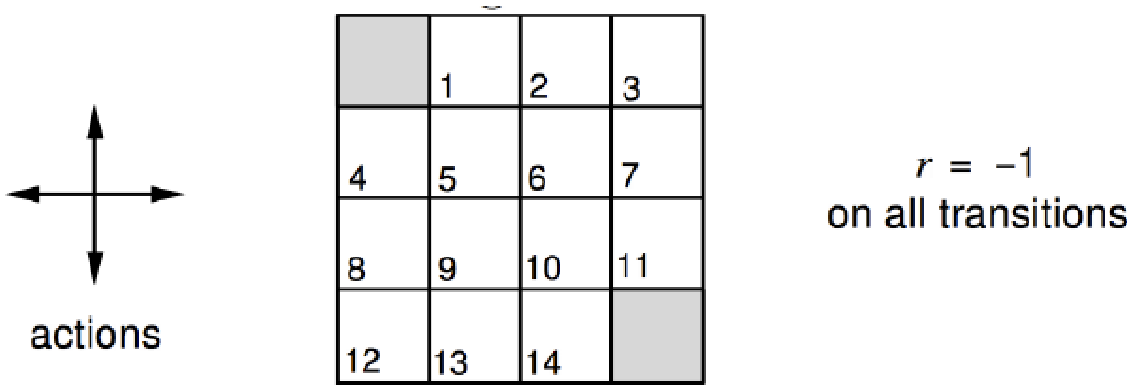
$$v_{k+1}(s) = \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s') \right)$$

Iterative Policy Evaluation

- Updating estimates v_{k+1} on the basis of other estimates v_k is often called bootstrapping.
- This leads to synchronous, full backups of the entire state space.
- Using synchronous backups, (synchronous: update all v values simultaneously)

- Convergence to v_π is guaranteed

Evaluating a Random Policy in the Small Gridworld



- Undiscounted episodic MDP (γ=1)
- Nonterminal states 1,…,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

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

Evaluating a Random Policy in the Small Gridworld

- Assume deterministic transition

$$\sum_{a\in A} P_{ss'}^{a} v_k(s') = v_k(s')$$

- Iterative Method

$$v_{k+1}(s) = \sum_{a\in A} \pi(a|s)\left(R_s^a + \gamma \sum_{s'\in S} P_{ss'}^{a} v_k(s')\right) = \sum_{a\in A} \pi(a|s)\left(-1 + v_k(s')\right)$$

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

$$v_{k+1}(s_1) = -1 + 0.25 * (0 + (-6.1) + (-8.4) + (-7.7)) = -6.55$$
$$v_{k+1}(s_6) = -1 + 0.25 * ((-7.7) + (-8.4) + (-8.4) + (-7.7)) = -9.05$$

Iterative Policy Evaluation in Small Gridworld

uniform policy

$k = 3$

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

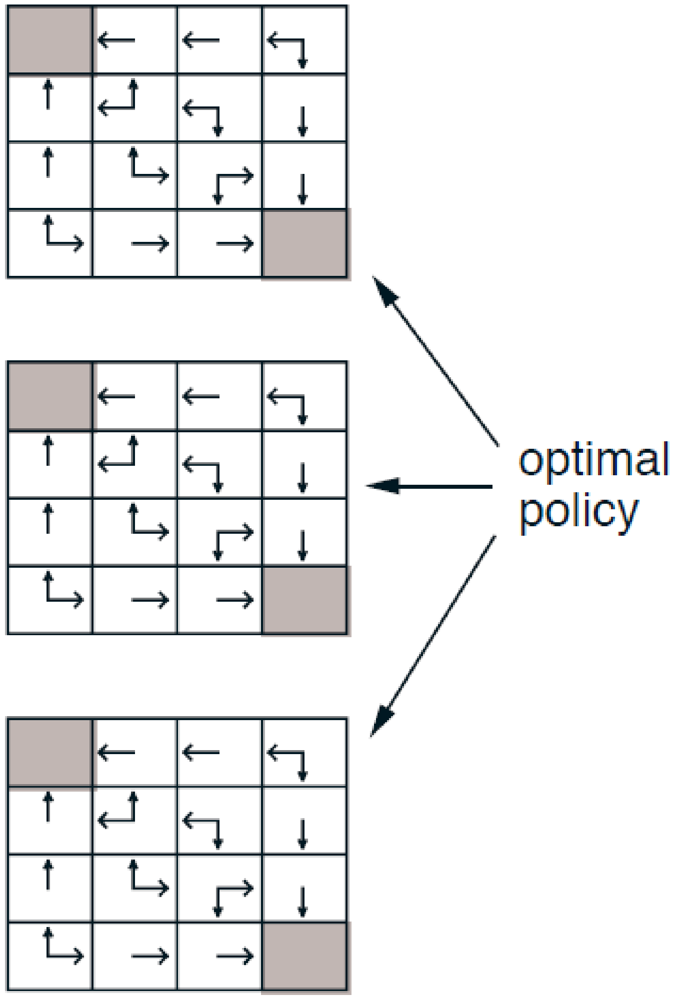
$k = 10$

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

greedy policy



R. Sutton and A. Barto, Reinforcement Learning: An Introduction, 2018

Control Problem

- We have seen how to evaluate a given policy
- However, our main goal is to find an optimal policy. How?

- Control problem:
- Policy Iteration
  - Policy Evaluation: given a policy, evaluate it (compute state-values)
  - Policy Improvement: improve policy

- Value Iteration
  - Compute state values (no policy is given)
  - At the end, derive optimal policy
  - No need of policy evaluation

Policy Iteration

- Policy Iteration
  1) Given an arbitrary policy π
  2) Evaluation: evaluate the policy π (= policy prediction)

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s') \right)$$

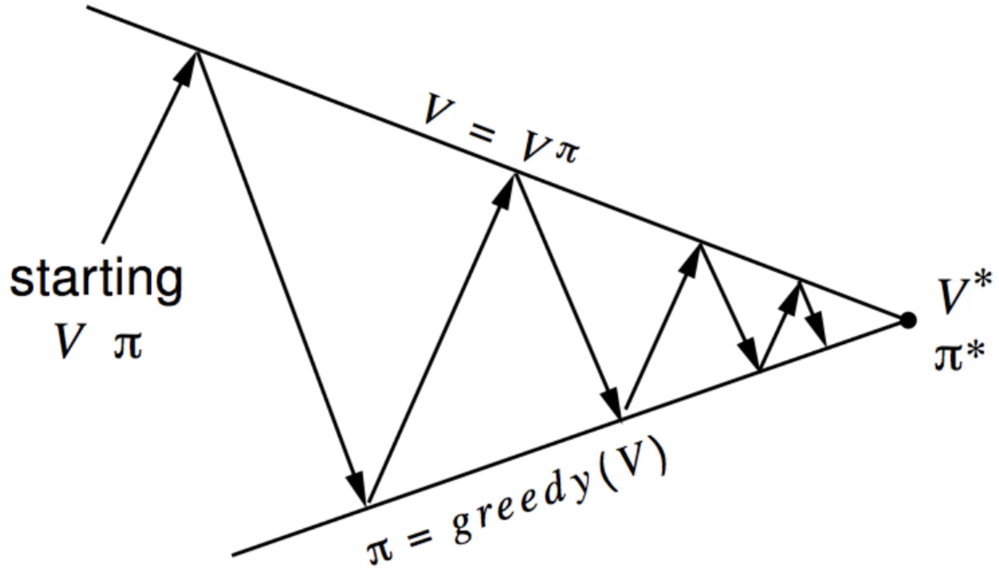
  3) Improvement: improve the policy by acting greedily with respect to v_π

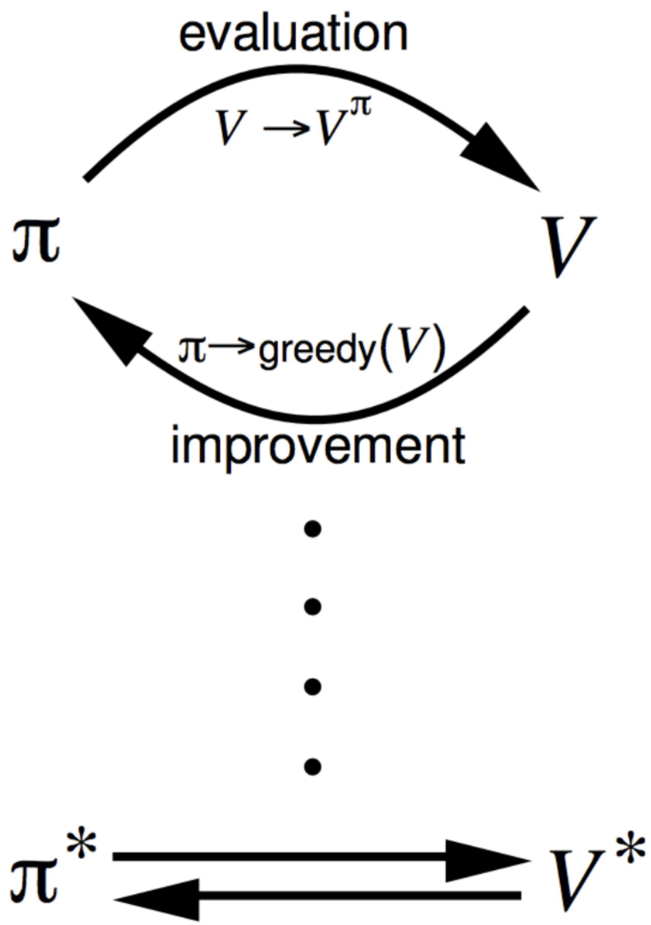
$$\pi' = greedy(v_\pi)$$

  4) We iterate these two processes 2)-3) until it converges

- Each policy evaluation step is fully executed, i.e. for each policy π_i an exact estimate of v_{π_i} is computed either by iterative method or by any other method

Policy Iteration





Policy evaluation Estimate v_π

Iterative policy evaluation

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

Greedy policy improvement

R. Sutton and A. Barto, Reinforcement Learning: An Introduction, 2018

Greedy Policy Improvement

- Consider a deterministic policy, $a = \pi(s)$ (Initial policy may be stochastic)
- Once policy evaluation is done, we can *improve* the policy by acting greedily
  - Select the best action according to $q(s_k, a_k)$ in every state

$$\pi'(s) = \mathrm{argmax}_{\mathrm{a}\in\mathrm{A}}\, q_\pi(s, a)$$

  - π' becomes a deterministic policy

- Is taking the greedy action $\pi'(s)$ is better than just following our policy π ?
  - This improves the value from any state s over one step

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

  - $\pi(s)$ is a deterministic policy

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

Greedy Policy Improvement

- **Policy Improvement Theorem**: Greedy policy improvement improves the value function, $v_{\pi'}(s) \geq v_\pi(s)$

Proof: $v_\pi(s) \leq q_\pi\big(s, \pi'(s)\big)$ (from previous page)

$$
\begin{aligned}
&= E_{\pi'}[R_{t+1} + \gamma v_\pi(s_{t+1}) | S_t = s] \\
&\leq E_{\pi'}\left[R_{t+1} + \gamma q_\pi\big(s_{t+1}, \pi'(s_{t+1})\big) \middle| S_t = s\right] \\
&\leq E_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi\big(s_{t+2}, \pi'(s_{t+2})\big) \middle| S_t = s\right] \\
&\leq E_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \cdots | S_t = s\right] = v_{\pi'}(s)
\end{aligned}
$$

Policy Improvement

- If improvements stop means, no changes in *q* or *v* values
  - Thus reached their maximum
  - Does it reach optimum policy?

$$q_\pi(s, \pi'(s)) \underset{\text{definition of } \pi'(s)}{=} \max_{a \in A} q_\pi(s, a) \underset{\text{no change}}{=} q_\pi(s, \pi(s)) \underset{\text{definition of } v_\pi(s)}{=} v_\pi(s)$$ (policy doesn't change)

- If the Bellman optimality equation has been satisfied

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

  - then $v_\pi(s) = v_*(s)$ for all $s \in S$
- So π is an optimal policy
- Policy improvement theorem guarantees finding optimal policies in finite MDPs

Modified Policy Iteration

- Does policy evaluation need to converge to v_π?
- Or should we introduce a stopping condition
  - e.g. ε-convergence of value function
- Or simply stop after *k* iterations of iterative policy evaluation?

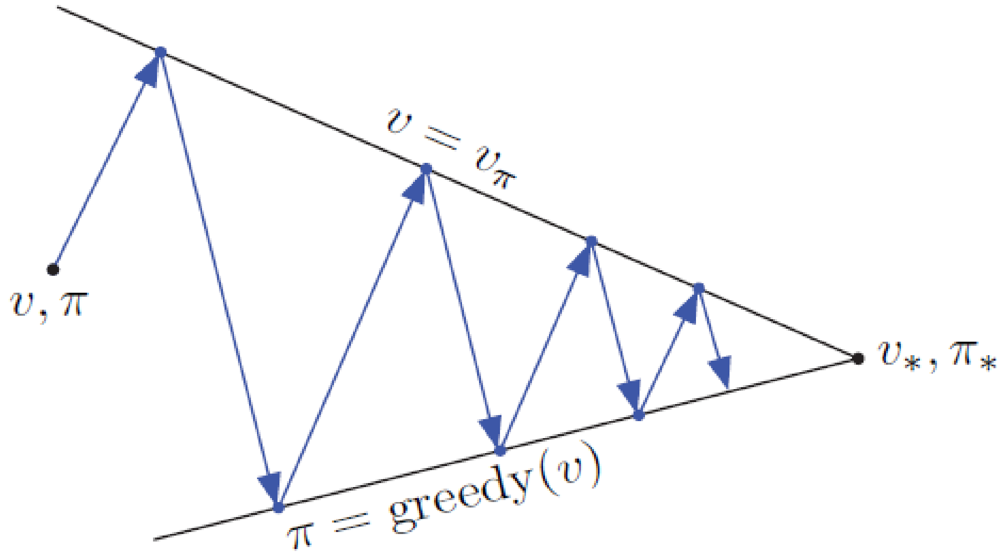
- For example, in the small grid-world *k=3* was sufficient to achieve optimal policy
- In value iteration, for example, only a *single iteration* of policy evaluation is performed in between each policy improvement.

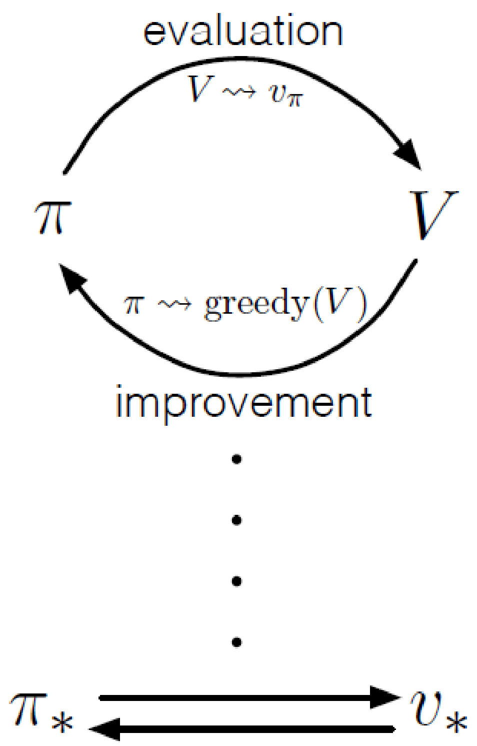
- As long as both processes continue to update all states, the ultimate result is typically the same
  - convergence to the optimal value function and an optimal policy.

Generalized Policy Iteration

- Generalized policy iteration(GPI) refers to all algorithms based on policy iteration
- Almost all reinforcement learning methods are well described as GPI.
- The policy always being improved with respect to the value function and the value function always being driven toward the value function for the policy
- If both the evaluation process and the improvement process stabilize, then the value function and policy must be optimal.
- Guaranteed to converge to the optimal policy, provided PE and PI are executed enough times.

Generalized Policy Iteration





Policy evaluation Estimate v_π

Any policy evaluation algorithm

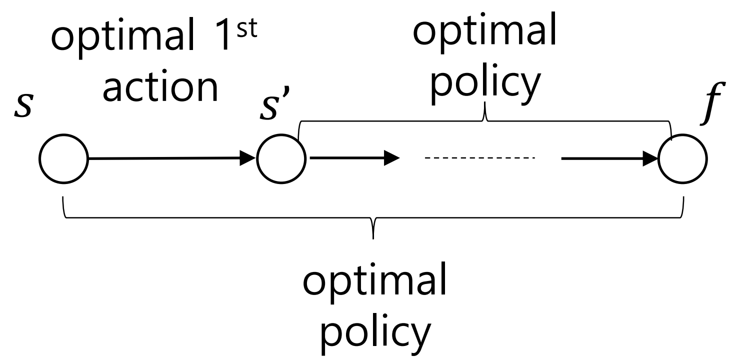
Policy improvement Generate $\pi' \geq \pi$

Any policy improvement algorithm

R. Sutton and A. Barto, Reinforcement Learning: An Introduction, 2018

Principle of Optimality

- Any optimal policy can be subdivided into two components:
  - An optimal 1st 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', $v_\pi(s') = v_*(s')$*

$$v_*(s) = \max_a \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')\right)$$

Value Iteration

- Policy iteration involves full policy evaluation steps between policy improvements.
- In large state-space MDPs, the full policy evaluation may be numerically very costly.
- Using a limited number of iterative policy evaluation steps and then apply policy improvement may speed up the entire DP process.

- Value iteration: the special case for Policy Iteration. The process of policy evaluation is stopped after one step.

Value Iteration

- Iterative application of Bellman optimality equation ($v_1 \rightarrow v_2 \rightarrow \ \ldots v_*$)
- Set k=1; Initialize $V_0(s) = 0$ for all states s
  - At each iteration k+1 [until converge]
    - For all states $s \in S$

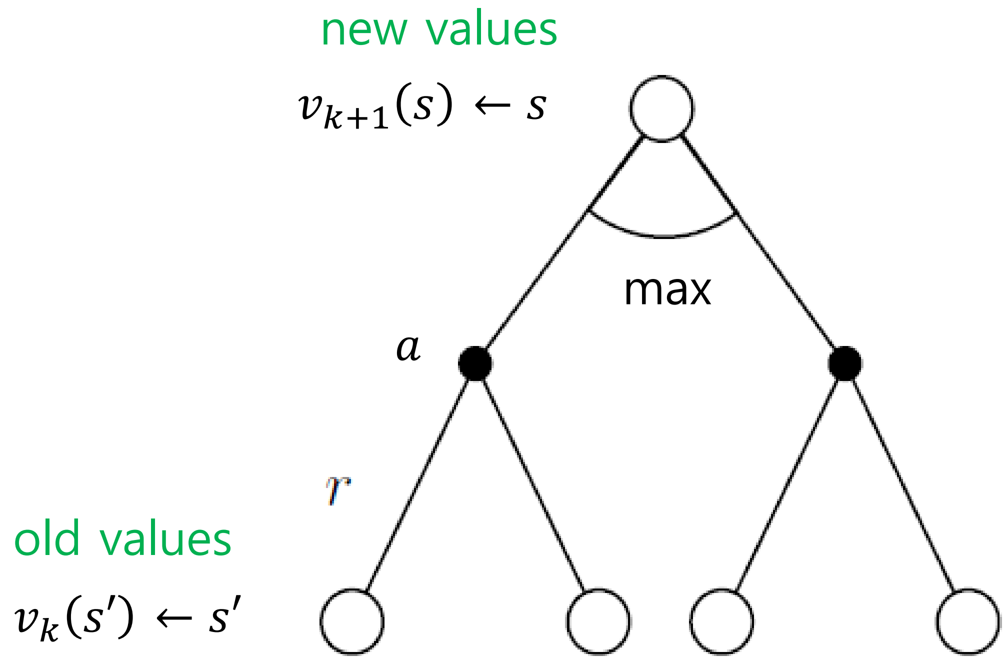
$$v_{k+1}(s) = \underbrace{\max_{a \in A}}_{\text{Policy improvement}} \Bigg(\underbrace{R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s')}_{\text{Policy evaluation}} \Bigg)$$

- Output: deterministic policy π such that

$$\pi_*(s) = \operatorname*{argmax}_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_{k+1}(s') \right)$$

- Convergences to v_*
- Unlike policy iteration, there is no explicit policy
- Intermediate value functions may not correspond to any policy

Value Iteration



Bellman Optimality Equation

$$v_*(s) = \max_a (R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s'))$$

Principle of Optimality

$$v_{k+1}(s) = \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s') \right)$$

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

Policy improvement + policy evaluation

Summarizing DP 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

- All DP Algorithms are based on state-value function $v_\pi(s)$ or $v_*(s)$
  - Complexity $O(mn^2)$ per iteration for *m* actions and *n* states
  - Evaluate all n^2 state transitions while considering up to *m* actions per state.
- Could also apply to action-value function $q_\pi(s,a)$ or $q_*(s,a)$
  - Complexity $O(mn^3)$ per iteration
  - There are up to nm action-values which require n^2 state transition evaluations each.

Asynchronous Dynamic Programming

- DP algorithms considered so far used synchronous backups:
  - In one iteration the entire state space is updated.
  - Computational expensive for large MDPs.
  - Some state-values or policy parts may converge faster than other but are updated as often as slowly converging states.
- Asynchronous backups update states individually in an (arbitrary) order:
  - Some states may be updated more frequently than others.
  - Choose smart order to achieve faster overall convergence rate.
  - Overall algorithms converges if all states are still visited to some extent
- Asynchronous DP backs up states individually, in any order
- For each selected state, apply the appropriate backup
- Can significantly reduce computation
- Guaranteed to converge if all states continue to be selected

Curse of Dimensionality

- DP uses full-width backups:
  - For each state update, every successor state and action is considered.
  - While utilizing full knowledge of the MDP structure.

- Hence, DP is can be effective up to medium-sized MDPs
- Also, we must have full knowledge of MDP structure

- For large problems DP suffers from the curse of dimensionality:
  - Number of finite states *n* grows exponentially with the number of state variables.
  - Also: if continuous variables need quantization typically a large number of states results.