Reinforcement Learning

Lecture 3

Dynamic Programming

Stergios Christodoulidis

MICS Laboratory
CentraleSupélec
Université Paris-Saclay

https://stergioc.github.io/

Last Lecture

Markov Process

- A Markov process is a memoryless random process
- A sequence of random states $[S_1, S_2, \dots]$ with the Markov property.

Definition

A Markov Process (or Markov Chain) is a tuple $< \mathcal{S}, \mathcal{P} >$

- S is a (finite) set of states
- P is a state transition matrix, i.e., $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$

Markov Reward Process

- A Markov reward process is a Markov chain with values.

Definition

A Markov Process (or Markov Chain) is a 4-tuple $< \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma >$

- S is a (finite) set of states
- P is a state transition matrix, i.e., $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$
- $\mathcal{R}$ is a reward function, $\mathcal{R}_s = \mathbb{E}[R_{t+1} \mid S_t = s]$
- γ is a discount factor, $\gamma \in [0, 1]$

Markov Decision Process

- A Markov decision process (MDP) is a Markov reward process with decisions. It is an environment in which all states are Markov

Definition

A Markov Process (or Markov Chain) is a 4-tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$

- $\mathcal{S}$ is a (finite) set of states
- $\mathcal{A}$ is a (finite) set of actions
- $\mathcal{P}$ is a state transition matrix, i.e., $\mathcal{P}_{ss'}^{a} = \mathbb{P}[S_{t+1} = s' \mid S_t = s,\ A_t = a]$
- $\mathcal{R}$ is a reward function, $\mathcal{R}_s^a = \mathbb{E}[R_{t+1} \mid S_t = s,\ A_t = a]$
- γ is a discount factor, $\gamma \in [0, 1]$

Bellman Expectation Equation

- The state-value function can again be decomposed into immediate reward plus discounted value of successor state,

$$\mathrm{v}_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma \mathrm{v}_\pi(S_{t+1}) \mid S_t = s]$$

- Similarly, the action-value can be decomposed as such,

$$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a]$$

Optimal Value Function

Definition

The optimal state-value function $v_*(s)$ *is the maximum value function over all policies*

$$v_*(s) = \max_\pi v_\pi(s)$$

The optimal action-value function $q_*(s,a)$ *is the maximum action-value function over all policies*

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

- The optimal value function specifies the best possible performance in the MDP.
- An MDP is “solved” when we know the optimal value fn.

Today’s Lecture

Today's Lecture

- Dynamic Programming
- Policy Evaluation
- Policy Iteration
- Value Iteration
- Extensions of DP (e.g., Asynchronous DP)

What is Dynamic Programming

- Dynamic: sequential or temporal component to the problem
- Programming: optimizing a "program", i.e., a policy

- A collection of algorithms to compute optimal policies/values given a known finite MDP.
- This is achieved by breaking them down into subproblems (Bellman Equations)
  - Solve the subproblems
  - Combine solutions of subproblems

Requirements for Dynamic Programming

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

Planning by Dynamic Programming

- Dynamic programming assumes full knowledge of the MDP, and It is used for planning:
- For Prediction:
  - Input: MDP/MRP and policy π
  - Output: State-value function
- For Control:
  - Input: MDP
  - Output: optimal value function and optimal policy

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)
- Bioinformatics (e.g., lattice models)

Policy Evaluation

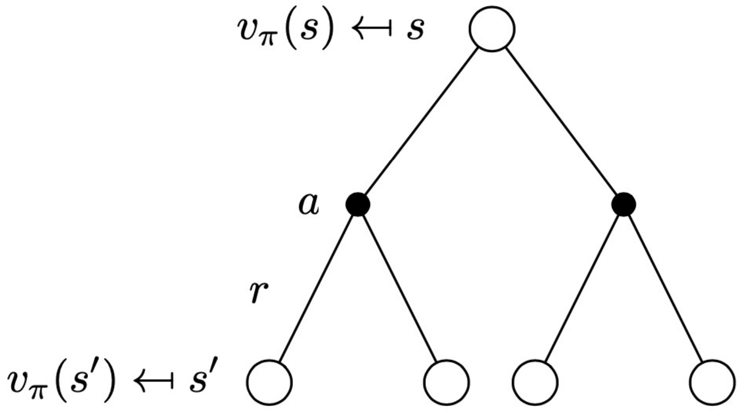
Iterative Policy Evaluation (1/3)

- Problem: evaluate a given policy π
- Solution: iterative application of Bellman expectation update

$$\mathrm{v}_1 \rightarrow \mathrm{v}_2 \rightarrow \cdots \rightarrow \mathrm{v}_\pi$$

- Using synchronous updates,
  - At each iteration $k + 1$
  - For all states $s \in \mathcal{S}'$
  - Update $\mathrm{v}_{k+1}(s)$ from $\mathrm{v}_k(s')$
  - where s' is a successor state of s
- We will discuss asynchronous updates later
- Convergence to v_π is proven at a geometric rate (refer to Banach's fixed-point theorem)

Iterative Policy Evaluation (2/3)



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

[An Introduction to Reinforcement Learning, Sutton and Barto]

Iterative Policy Evaluation (3/3)

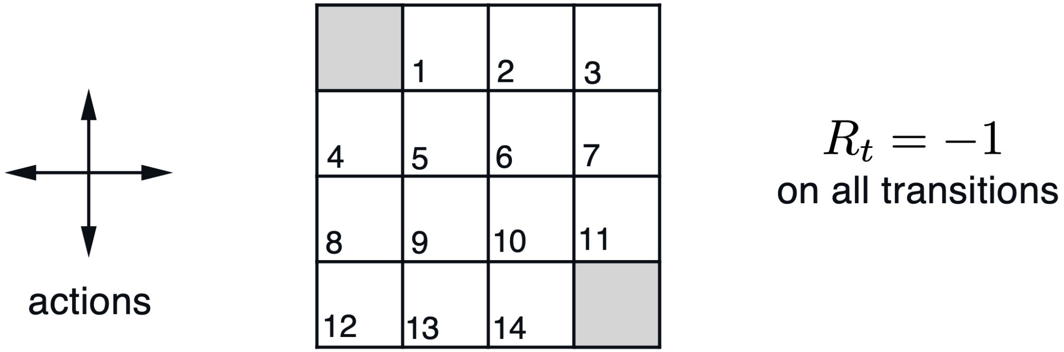
Iterative Policy Evaluation, for estimating $V \approx v_\pi$

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

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

[An Introduction to Reinforcement Learning, Sutton and Barto]

Evaluating a Random Policy in the Small Gridworld



- Undiscounted episodic MDP ($\gamma = 1$)
- Nonterminal states 1, ..., 14
- Terminal states shown 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

$$\pi(u \mid \cdot) = \pi(d \mid \cdot) = \pi(r \mid \cdot) = \pi(l \mid \cdot) = 0.25$$

[An Introduction to Reinforcement Learning, Sutton and Barto]

Iterative Policy Evaluation in Small Gridworld (1/2)

v_k for the random 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

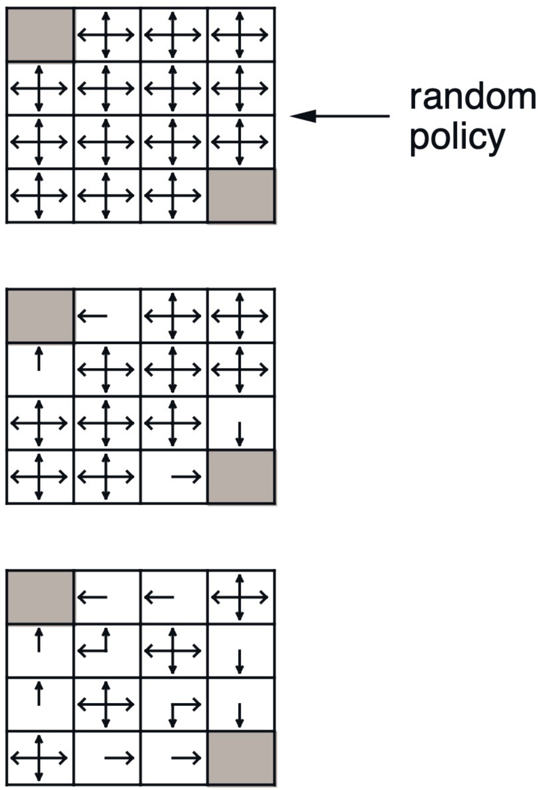
$k = 1$

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

greedy policy w.r.t. v_k



[An Introduction to Reinforcement Learning, Sutton and Barto]

Iterative Policy Evaluation in Small Gridworld (2/2)

$k = 3$

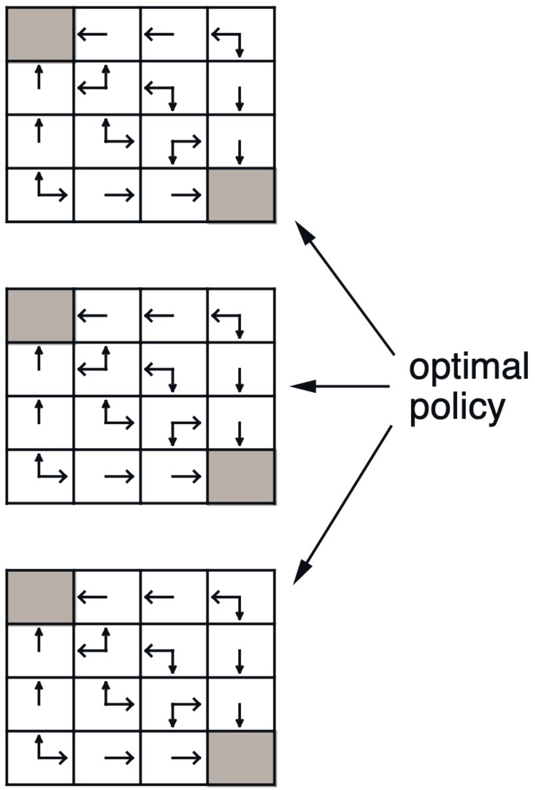
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

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



[An Introduction to Reinforcement Learning, Sutton and Barto]

Policy Iteration

How to Improve a Policy

- Given a policy π
  - Evaluate the policy π

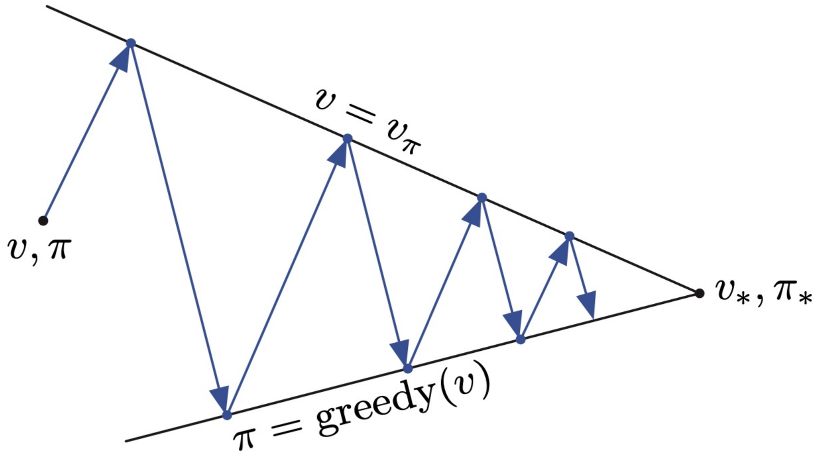
$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s]$$

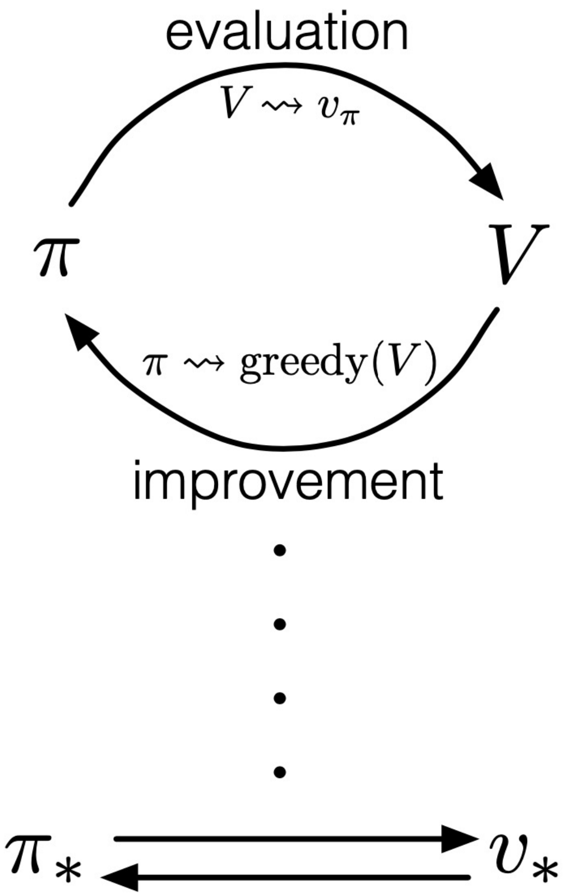
  - Improve the policy by acting greedily with respect to v_π

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

- 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 π∗

Policy Iteration





- Policy evaluation Estimate v_π
  - Iterative policy evaluation
- Policy improvement Generate $\pi' \geq \pi$
  - Greedy policy improvement

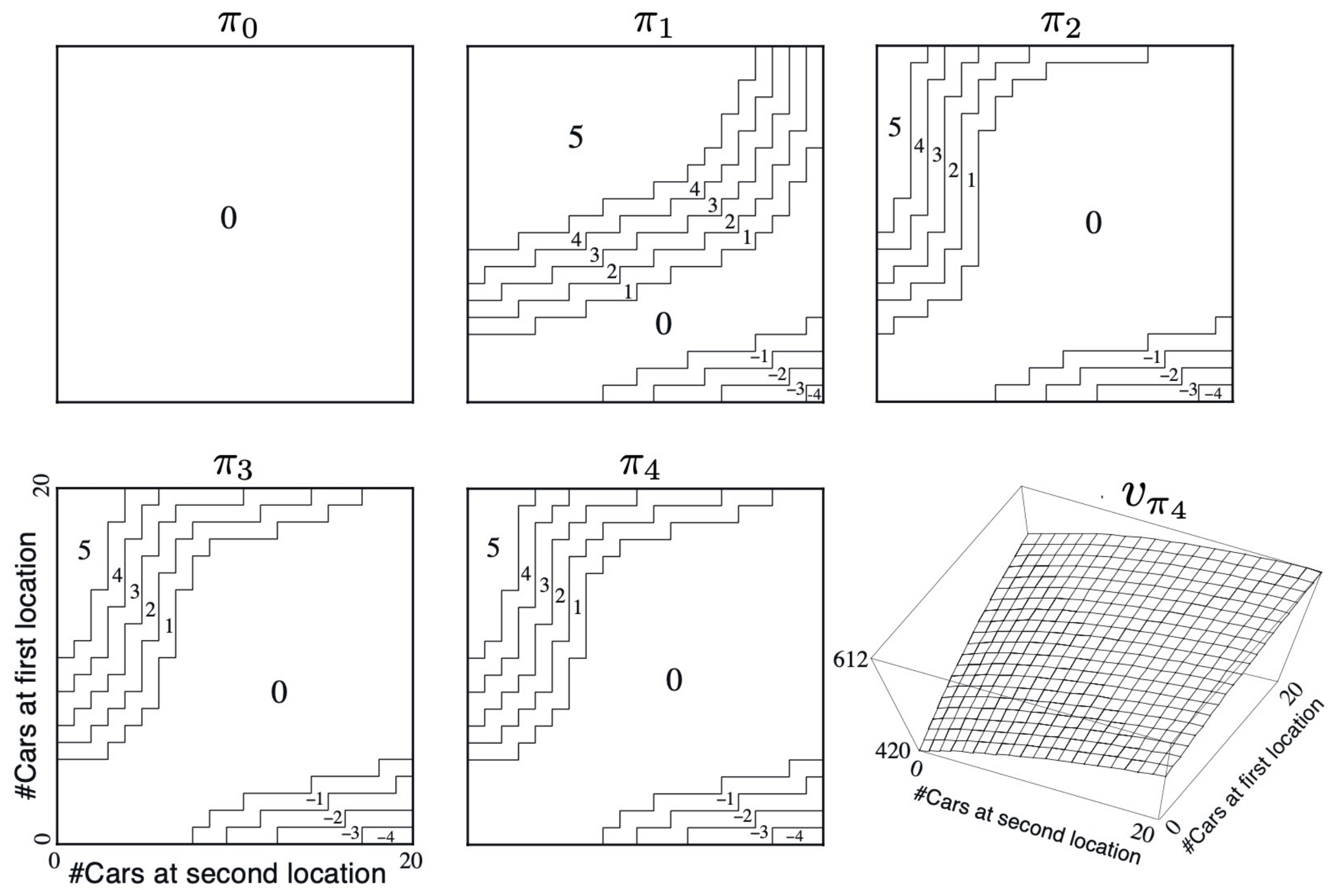
[An Introduction to Reinforcement Learning, Sutton and Barto]

Jack's Car Rental Problem

- States: Two locations, maximum of 20 cars at each
- Actions: Move up to 5 cars between locations overnight
- Reward: $10 for each car rented (must be available)
- Transitions: Cars returned and requested randomly
  - Poisson distribution, n returns/requests with probability $\frac{\lambda^n}{n!}e^{-\lambda}$
  - 1st location: average requests = 3, average returns = 3
  - 2nd location: average requests = 4, average returns = 2

[Google Maps, West Edmonton Mall. Canada]

Policy Iteration in Jack's Car Rental



Policy Improvement (1/3)

- Consider a deterministic policy, $a = \pi(s)$
- We can improve the policy by acting greedily

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

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

$$q_{\pi}\left(s, \pi^{'}(s)\right) = \max_{a \in \mathcal{A}} q_{\pi}(s, a) \geq q_{\pi}(s, \pi(s)) = \mathrm{v}_{\pi}(s)$$

- It therefore improves the value function, $\mathrm{v}_{\pi'}(s) \geq \mathrm{v}_{\pi}(s)$
    - $\pi_0 \rightarrow (\mathrm{Eva}l) \rightarrow \mathrm{v}_0 \rightarrow (\mathrm{I}mp) \rightarrow \pi_1 \rightarrow (\mathrm{Eval}) \rightarrow \mathrm{v}_1 \ldots \rightarrow \mathrm{v}_{\pi}$

Policy Improvement (2/3)

- If improvements stop,

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

- Then the Bellman optimality equation has been satisfied!

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

- Therefore, $\mathrm{v}_\pi(s) = \mathrm{v}_*(s)$ for all $s \in \mathcal{S}$
- So π is an optimal policy

Policy Iteration (3/3)

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 $s \in \mathcal{S}$; $V(terminal) \doteq 0$

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

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

[An Introduction to Reinforcement Learning, Sutton and Barto]

Value Iteration

Principle of Optimality

- 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 for state s, $\mathrm{v}_\pi(s) = \mathrm{v}_*(s)$, if and only if

- *For any state s' reachable from s*
- *π achieves the optimal value from state s',* $\mathrm{v}_\pi(s') = \mathrm{v}_*(s')$

Deterministic Value Iteration

- If we know the solution to subproblems $\mathrm{v}_*(s')$
- The solution $\mathrm{v}_*(s)$ can be found by one-step lookahead

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

- The idea of value iteration is to apply these updates iteratively
- Intuition: start with final rewards and work backwards
- Still works with loopy, stochastic MDPs

Example: Shortest Path

g			

Problem

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

V_1

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

[David Silver, IRL, UCL 2015]

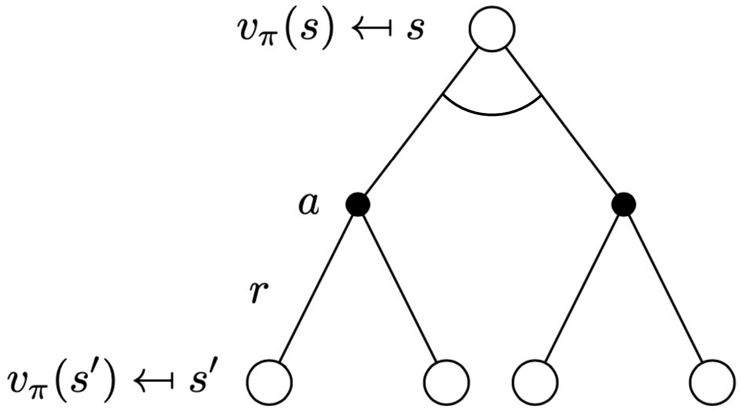
Value Iteration (1/2)

- Problem: find optimal policy π
- Solution: iterative application of Bellman optimality update

$$\mathrm{v}_1 \rightarrow \mathrm{v}_2 \rightarrow \cdots \rightarrow \mathrm{v}_*$$

- Using synchronous updates
  - At each iteration k + 1
  - For all states $s \in \mathcal{S}$
  - Update $\mathrm{v}_{k+1}(s)$ from $\mathrm{v}_k(s')$
- Unlike policy iteration, there is no explicit policy
- Intermediate value functions may not correspond to any policy

Value Iteration (2/2)



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

[An Introduction to Reinforcement Learning, Sutton and Barto]

Synchronous Dynamic Programming Algorithms

Problem	Bellman Equation	Algorithm
Prediction	Bellman Expectation Equation	Iterative Policy Evalution
Control	Bellman Expectation Equation + Greedy Policy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

- Algorithms are based on state-value function $\mathrm{v}_{\pi}(s)$ or $\mathrm{v}_{*}(s)$
- Complexity $O(mn^2)$ per iteration, for m actions and n states

Asynchronous DP

Asynchronous Dynamic Programming

- DP methods described so far used synchronous updates
- i.e. all states are updated in parallel
- Asynchronous DP updates states individually, in any order
- For each selected state, apply the appropriate update
- Can significantly reduce computation
- Guaranteed to converge if all states continue to be selected

Asynchronous Dynamic Programming

- Some simple ideas for asynchronous dynamic programming:
  - In-place dynamic programming
  - Prioritised sweeping

In-Place Dynamic Programming

- Synchronous value iteration stores two copies of value function for all s in $\mathcal{S}$

$$\mathrm{v}_{\text{new}}(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \mathrm{v}_{\text{old}}(s') \right) \text{, at the end } \mathrm{v}_{old}(s) \leftarrow \mathrm{v}_{\text{new}}(s)$$

- In-place value iteration only stores one copy of value function for all s in $\mathcal{S}$

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

Prioritised Sweeping

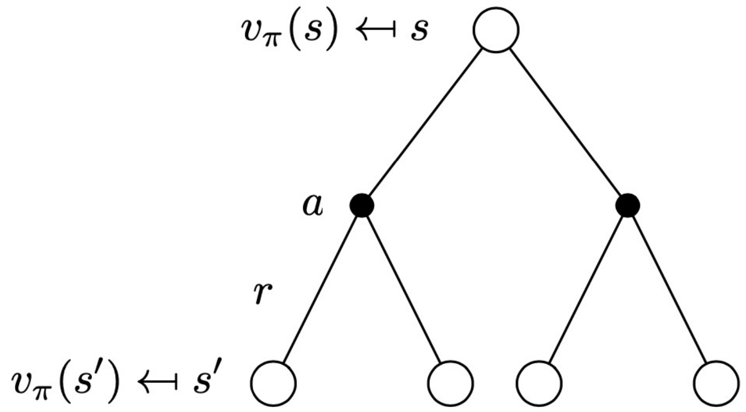
- Use magnitude of Bellman error to guide state selection, e.g.

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

- 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

Full-Width Updates

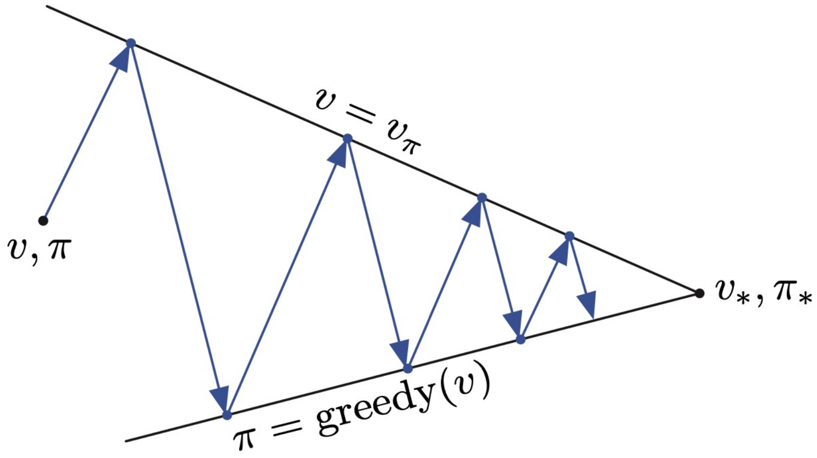
- DP uses full-width backups
- For each backup (sync or async)
  - Every successor state and action is considered
  - Using knowledge of the MDP transitions and reward function
- DP is effective for medium-sized problems (millions of states)
- For large problems DP suffers Bellman's curse of dimensionality
  - Number of states n = |S| grows exponentially with number of state variables
- Even one backup can be too expensive

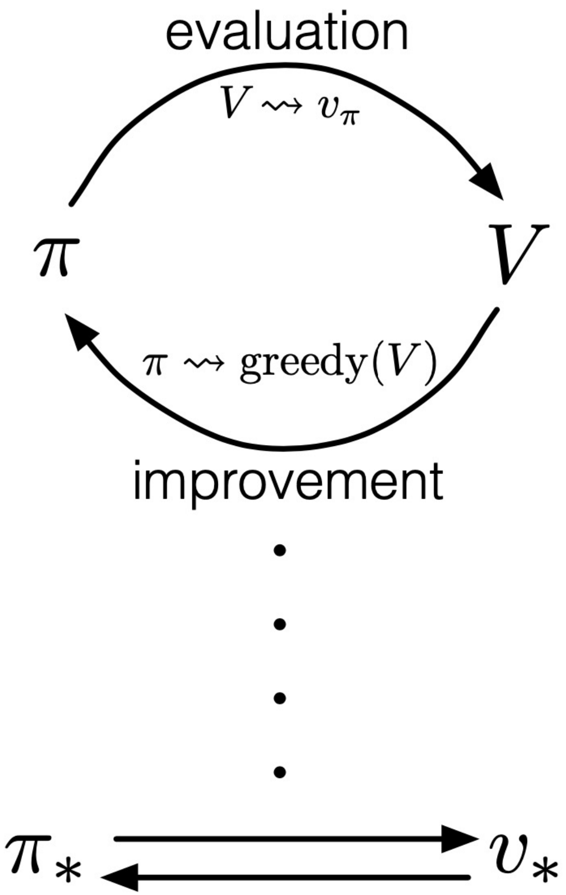


Sample Updates

- In subsequent lectures we will consider sample backups
- Using sample rewards and sample transitions
- Instead of reward function R and transition dynamics P
- Advantages:
    - Model-free: no advance knowledge of MDP required
    - Breaks the curse of dimensionality through sampling
    - Cost of backup is constant, independent of $n = |S|$

Generalized Policy Iteration





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

[An Introduction to Reinforcement Learning, Sutton and Barto]

Summary

- Policy evaluation refers to the iterative computation of the value function for a give policy
- Policy improvement refers to the computation of an improved policy given the value function
- Putting these two together:
  - Policy Iteration
  - Value Iteration
- These iterative updates are closely related to the Bellman equations (Bellman equations turned into assignment statements)
- When the updates no longer results in new values -> Convergence (Optimality)
- No need to sweep all states, asynchronous DP.