Lecture 9: Dynamic Programming

Ziyu Shao

School of Information Science and Technology
ShanghaiTech University

May 04 & 06, 2020

Outline

Outline

What is Dynamic Programming?

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

- c.f. linear programming

- A method for solving complex problems
- By breaking them down into subproblems
  - Solve the subproblems
  - Combine solutions to 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

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)

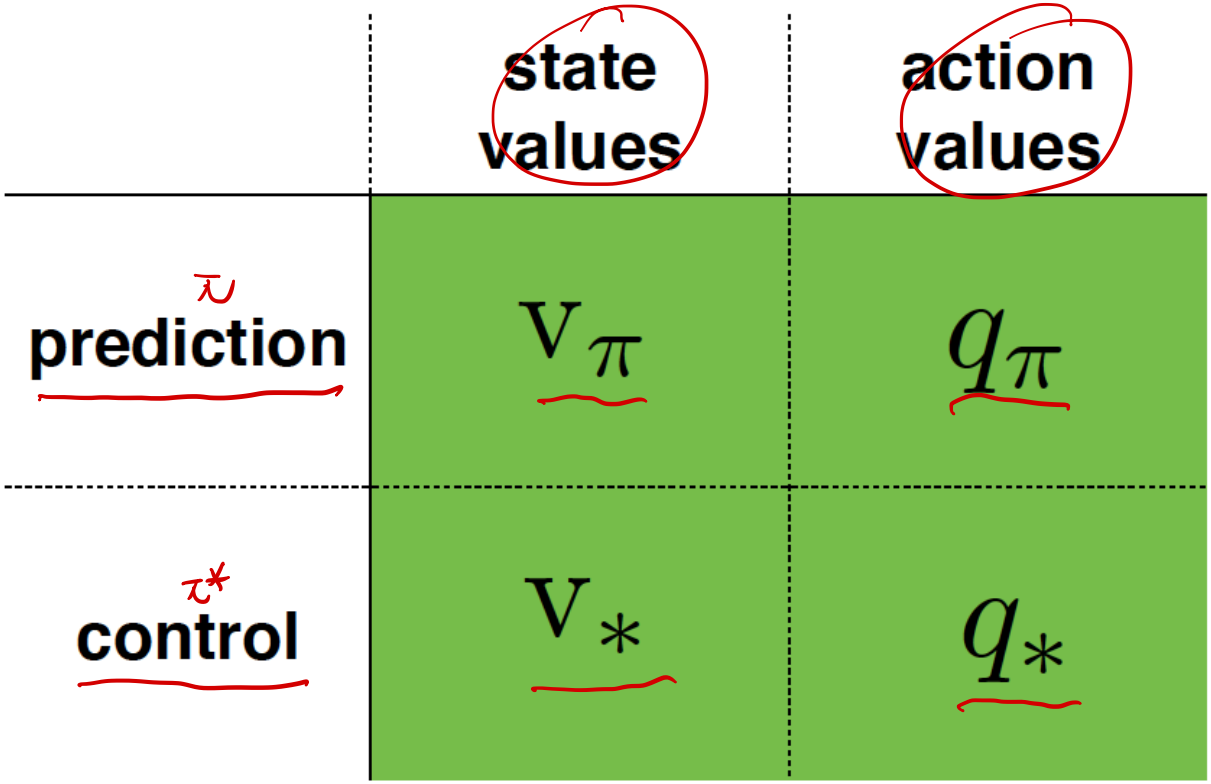
Recall: Definitions

- Agent: an entity that is equipped with sensors, in order to sense the environment, and end-effectors in order to act in the environment, and goals that he wants to achieve
- Policy: a mapping function from observations (sensations, inputs of the sensors) to actions of the end effectors
- Model: the mapping function from states/observations and actions to future states/observations
- Planning: unrolling a model forward in time and selecting the best action sequence that satisfies a specific goal
- Plan: a sequence of actions

Recall: Definitions

- Rollout: several sequential transitions
- Trajectory: full stack of observed quantities of sequences of states, actions and rewards

Recall: Four Basic Value Functions



Bellman Expectation Equation

Bellman optimality Equation.

Planning by Dynamic Programming

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

Bellman Backup

- The term "Bellman backup" comes up quite frequently in the RL literature.
- The Bellman backup for a state (or a state-action pair) is the right-hand side of the Bellman equation: the reward-plus-next-value.
- It is a particular computation of calculating a new value based on successor-values
- For example, a Bellman backup at state s with respect to a value function V computes a new value at s by backing up the successor values $V(s')$ using Bellman equation

Categories of Policies

- First Criterion
  - Non-stationary policy: depends on the time step & useful for the finite-horizon problem
  - Stationary policy: independent of the time step & useful for the infinite-horizon problem
- Second Criterion
  - Deterministic policy: described as $\pi(s) : \mathcal{S} \rightarrow \mathcal{A}$
  - Stochastic policy: described as $\pi(a|s) : \mathcal{S}x\mathcal{A} \rightarrow [0, 1]$ where $\pi(a|s)$ denotes the probability that action a may be chosen in state s

Outline

Iterative Policy Evaluation

- Problem: evaluate a given policy π
- Solution: iterative application of Bellman expectation backup
- $v_1 \rightarrow v_2 \ldots \rightarrow v_\pi$
- Using *synchronous* backups,
  - At each iteration $k+1$
  - For all states $s \in \mathcal{S}$
  - Update $v_{k+1}(s)$ from $v_k(s')$
  - 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

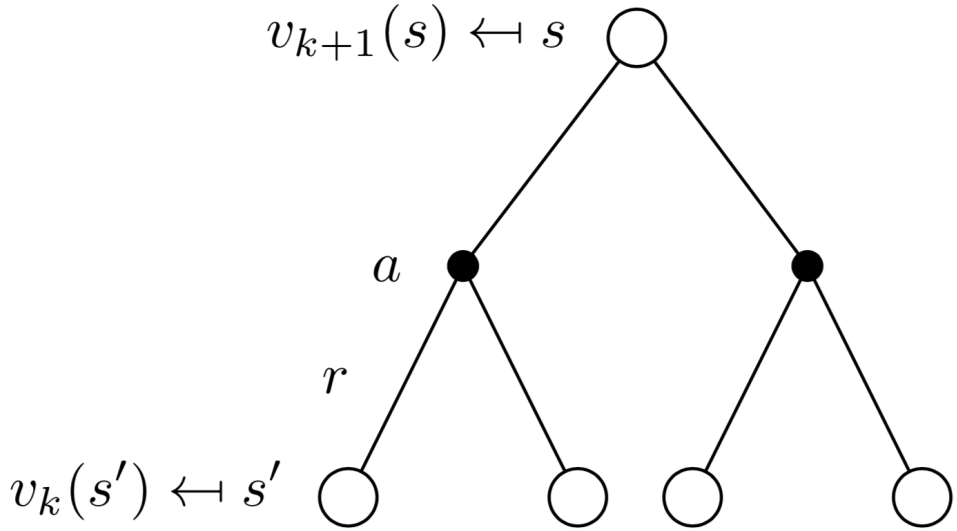
Iterative Policy Evaluation: Sweep

$$v_0 \to v_1 \to \cdots \to v_k \to v_{k+1} \to \cdots \to v_\pi$$

a "sweep"

A sweep consists of applying a **backup operation** to each state.

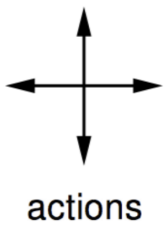
Iterative Policy Evaluation: Backup



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

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

Example: Small Gridworld



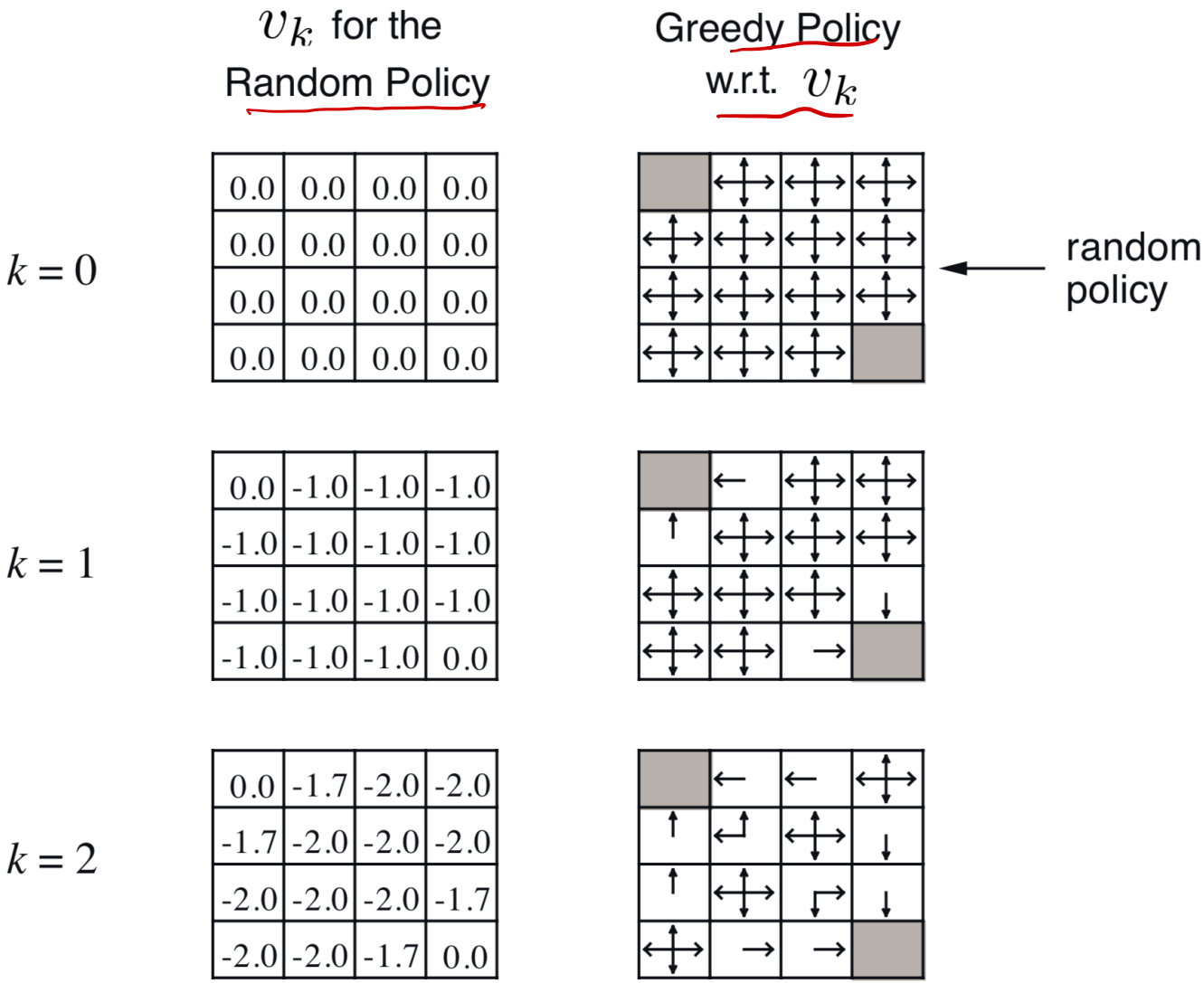
	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

$r = -1$
on all transitions

- Undiscounted episodic MDP ($\gamma = 1$)
- 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

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

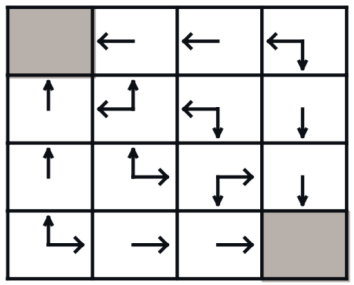
Iterative Policy Evaluation in Small Gridworld



Iterative Policy Evaluation in Small Gridworld (2)

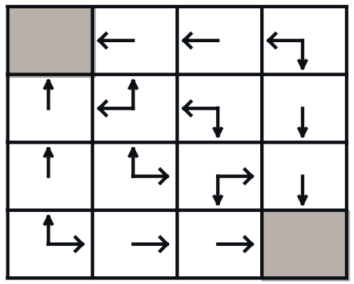
$k = 3$

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



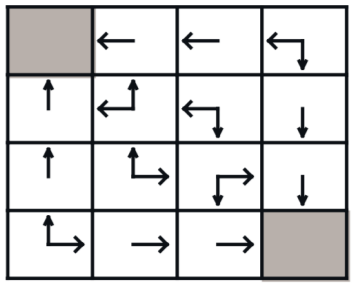
$k = 10$

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



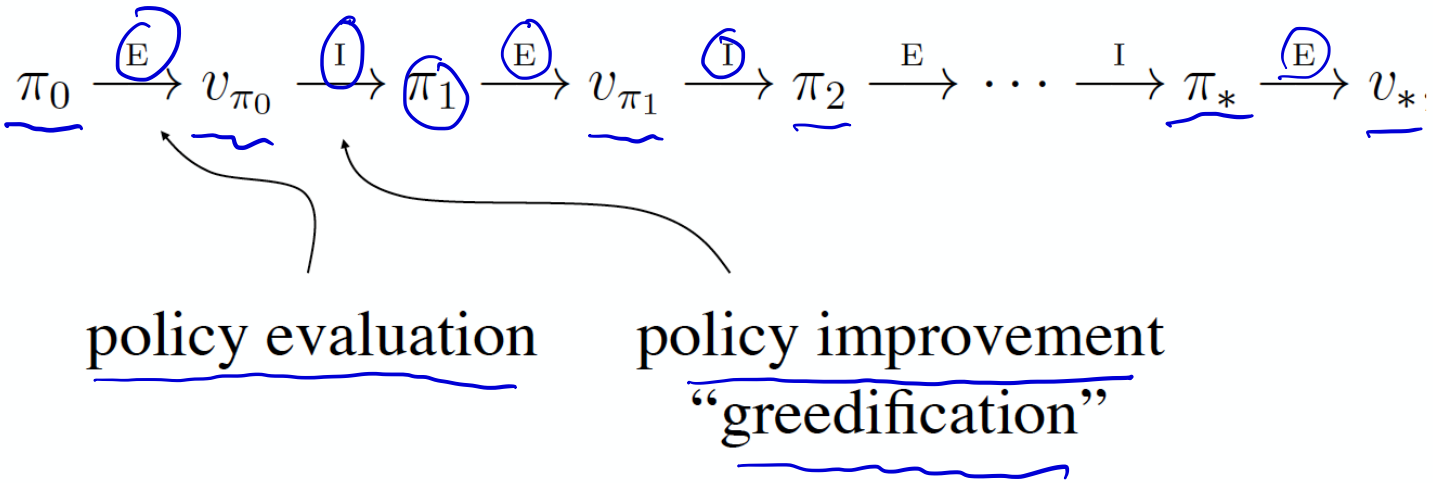
optimal policy

Outline

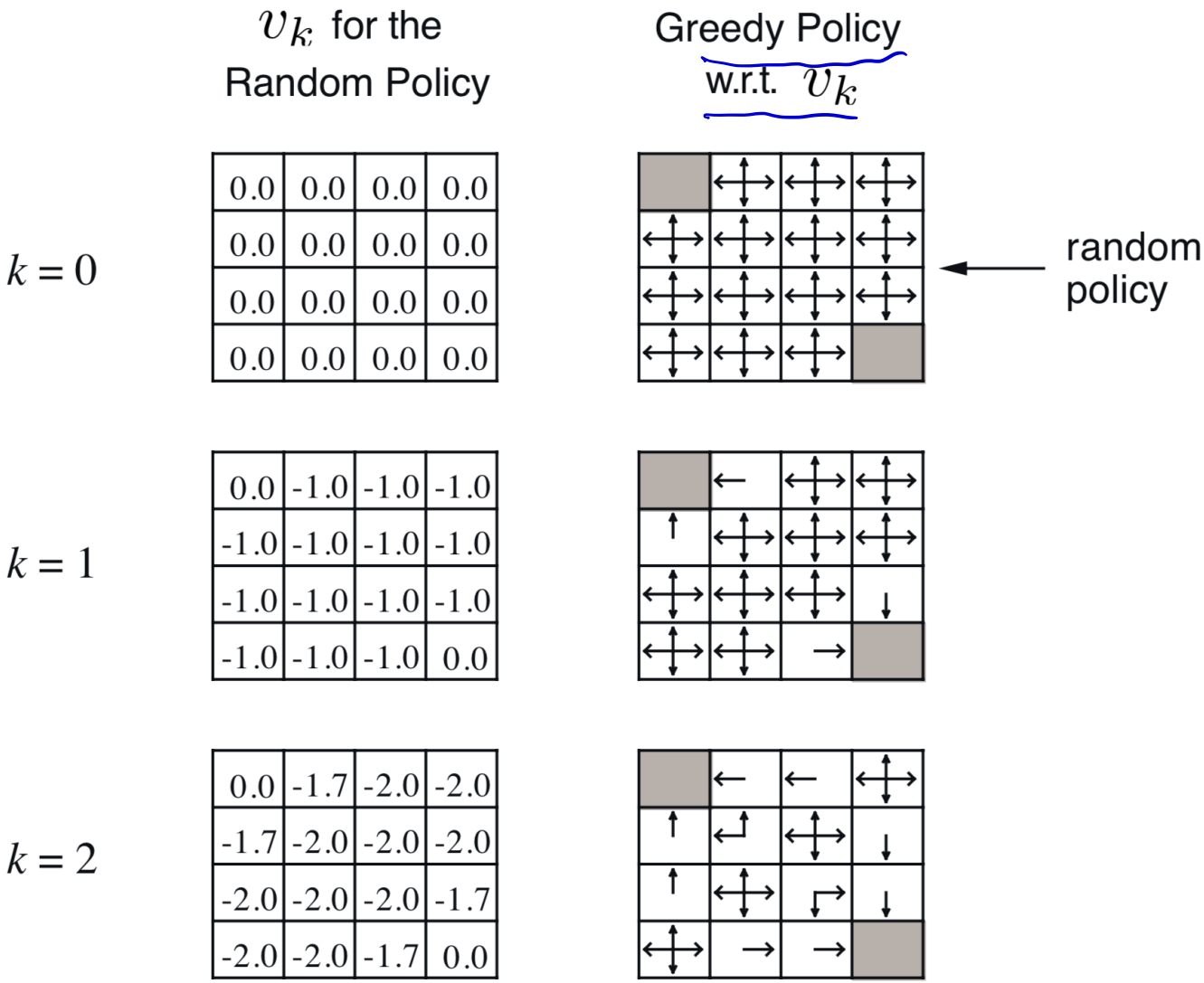
prediction

Control.

Policy Iteration



Policy Improvement in Small Gridworld



Policy Improvement

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

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

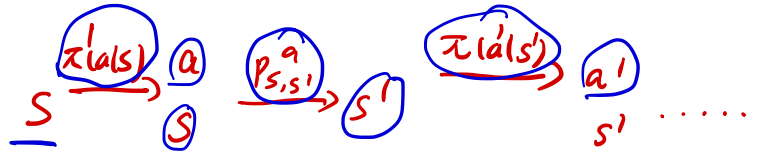
- This improves the value from any state s over one step,

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

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

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

Remark & Proof



① $E_{\pi'}[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s]$: the expected discounted reward value when starting in state s, choosing actions according to π' for the next step, and then according to π thereafter.

$= E[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s, A_t = \pi'(s)]$

② $E_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 V_\pi(S_{t+2}) | S_t = s]$: the expected discount reward value when starting in state s, choosing actions according to π' for the next two steps, and then according to π thereafter.

Remark & Proof

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

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

(1) $v_\pi(s) \leq q_\pi(s, \pi'(s)) \triangleq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = \pi'(s)]$

$= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]$

$\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s]$

$= \mathbb{E}_{\pi'}[R_{t+1} \mid S_t = s] + \gamma \mathbb{E}_{\pi'}[q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s]$

$v_\pi(s) \leq q_\pi(s, \pi'(s))$

$v_\pi(S_{t+1}) \leq q_\pi(S_{t+1}, \pi'(S_{t+1}))$

Since $q_\pi(S_t, \pi'(S_t)) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t, A_t = \pi'(S_t)]$

$\Rightarrow q_\pi(S_{t+1}, \pi'(S_{t+1})) = \mathbb{E}[R_{t+2} + \gamma v_\pi(S_{t+2}) \mid S_{t+1}, A_{t+1} = \pi'(S_{t+1})]$

$= \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2}) \mid S_{t+1}]$

$\Rightarrow \mathbb{E}_{\pi'}[q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s] = \mathbb{E}_{\pi'}[\mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2}) \mid S_{t+1}] \mid S_t = s]$

Adam + Markov $= \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2}) \mid S_t = s]$

Remark & Proof

② $V_\pi(s) \le E_{\pi'}[R_{t+1} | S_t = s] + \gamma E_{\pi'}[R_{t+2} + \gamma V_\pi(S_{t+2}) | S_t = s]$

$= E_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 V_\pi(S_{t+2}) | S_t = s]$

③ Therefore, we have

$V_\pi(s) \le E_{\pi'}[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s]$

$\le E_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 V_\pi(S_{t+2}) | S_t = s]$

$\le E_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 V_\pi(S_{t+3}) | S_t = s]$

by recursive induction.

$\le E_{\pi'}[\underbrace{R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots}_{G_t} | S_t = s]$

$= E_{\pi'}[G_t | S_t = s]$

$= V_{\pi'}(s)$

π' is improving !

Remark & Proof

Policy Improvement (2)

Another Perspective in later section.

- If improvements stop, $\pi' = \pi$

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

- Then the Bellman optimality equation has been satisfied

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

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

Policy Iteration

- Given a policy π
  - Evaluate the policy π

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

  - Improve the policy by acting greedily with respect to v_π

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

- 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 Algorithm

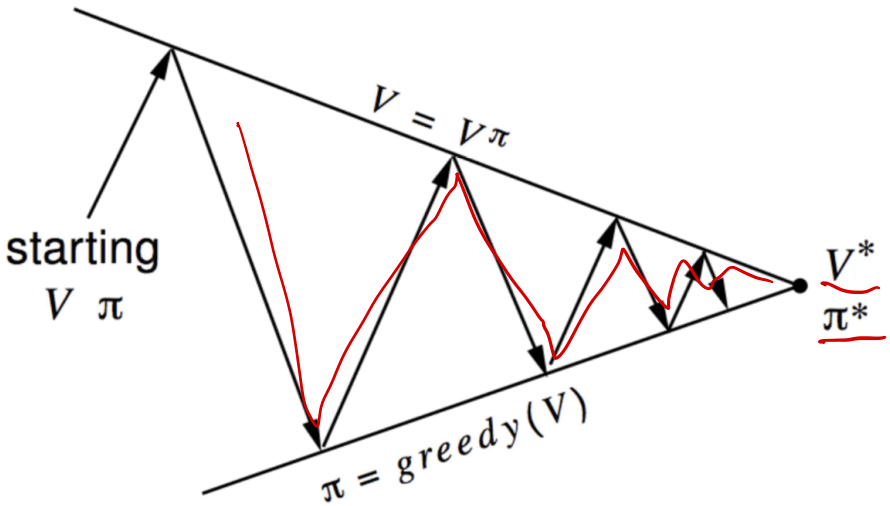
Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

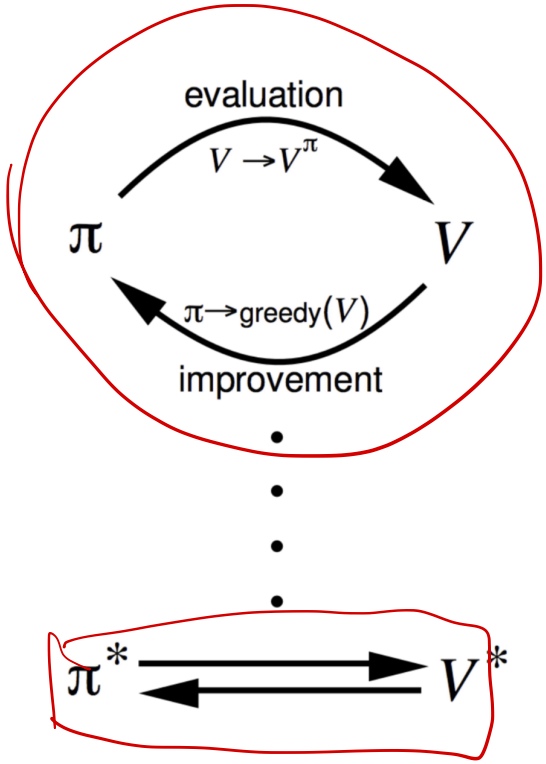
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   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 $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
      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

Policy Iteration





Policy evaluation Estimate v_π

Iterative policy evaluation

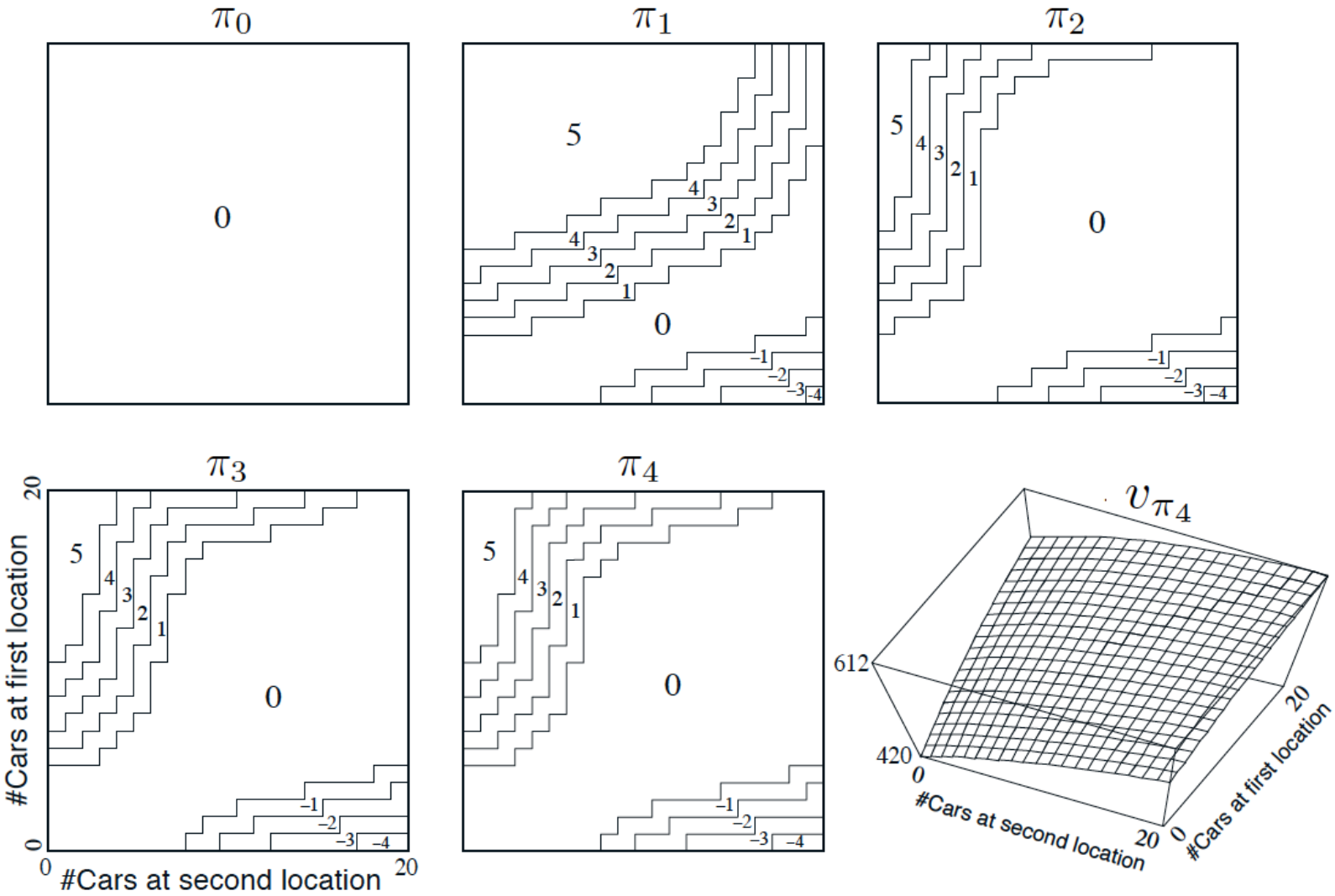
Policy improvement Generate $\pi' \geq \pi$

Greedy policy improvement

Example: Jack's Car Rental

- 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 prob $\frac{\lambda^n}{n!}e^{-\lambda}$
  - 1st location: average requests $= 3$, average returns $= 3$
  - 2nd location: average requests $= 4$, average returns $= 2$

Policy Iteration in Jack's Car Rental



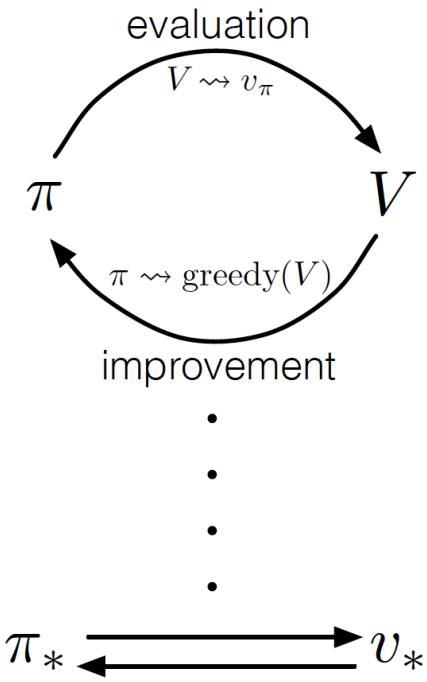
Modified Policy Iteration

- 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 gridworld $k = 3$ was sufficient to achieve optimal policy
- Why not update policy every iteration? i.e. stop after $k = 1$
  - This is equivalent to *value iteration* (next section)

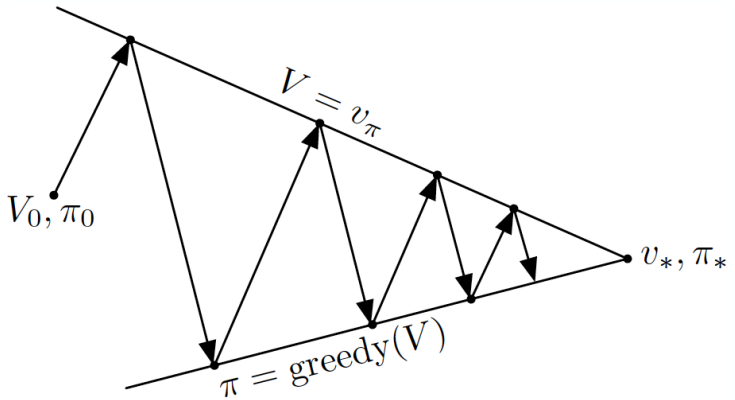
Generalized Policy Iteration (GPI)

Actor-critic

- GPI: interaction of Policy Evaluation and Policy Improvement
- Policy Evaluation estimate v_π: any policy evaluation algorithm
- Policy Improvement generate $\pi' \geq \pi$: any policy improvement algorithm



A geometric metaphor for convergence of GPI:



Outline

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 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', $v_\pi(s') = v_*(s')$*

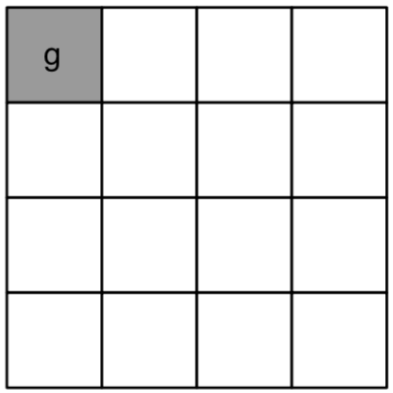
Deterministic Value Iteration

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

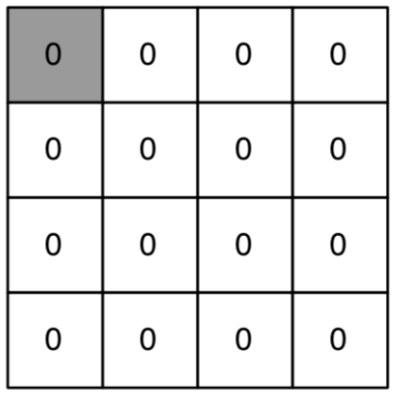
$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \left\{ \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right\}$$

- 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

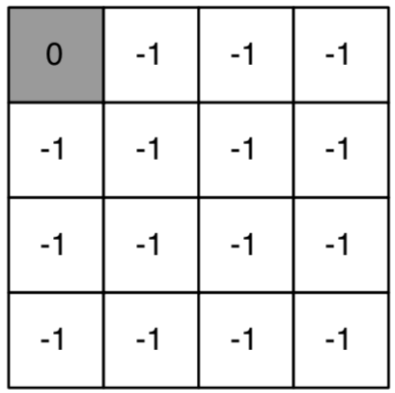
Example: Shortest Path



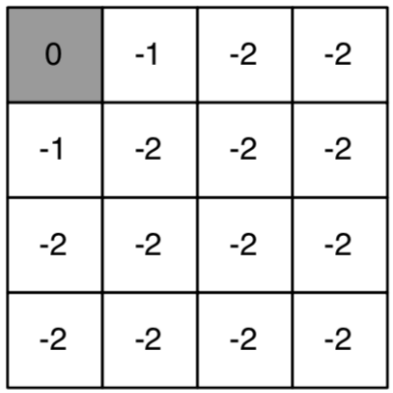
Problem



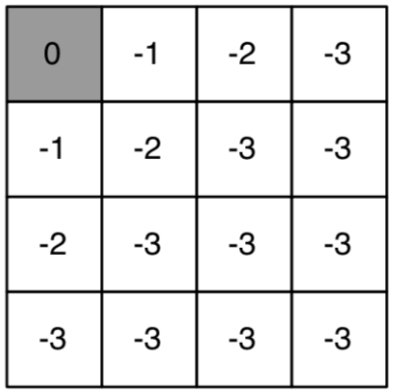
V_1



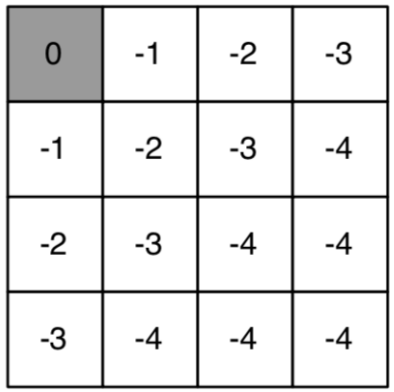
V_2



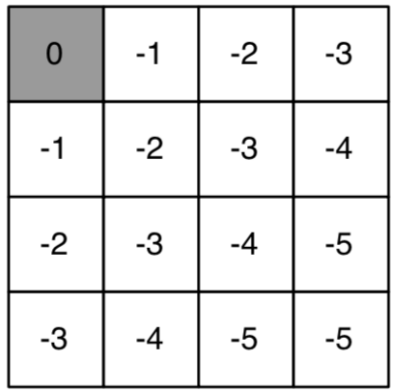
V_3



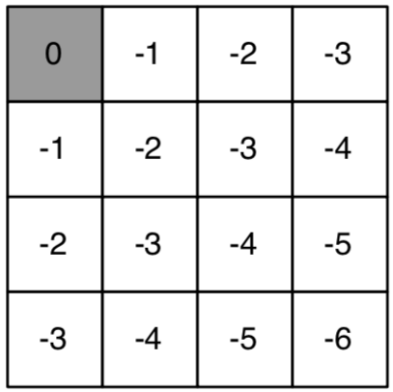
V_4



V_5



V_6



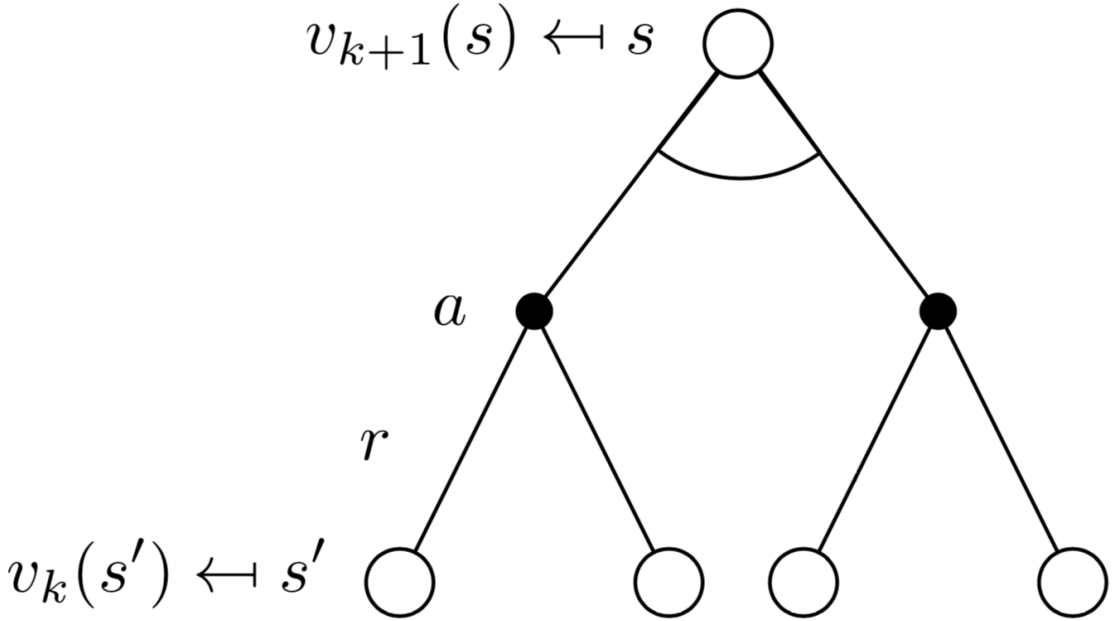
V_7

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
  - 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

$\arg\max_a q_*(s,a)$

Value Iteration (2)



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

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

Value Iteration Algorithm

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 $V(terminal) = 0$

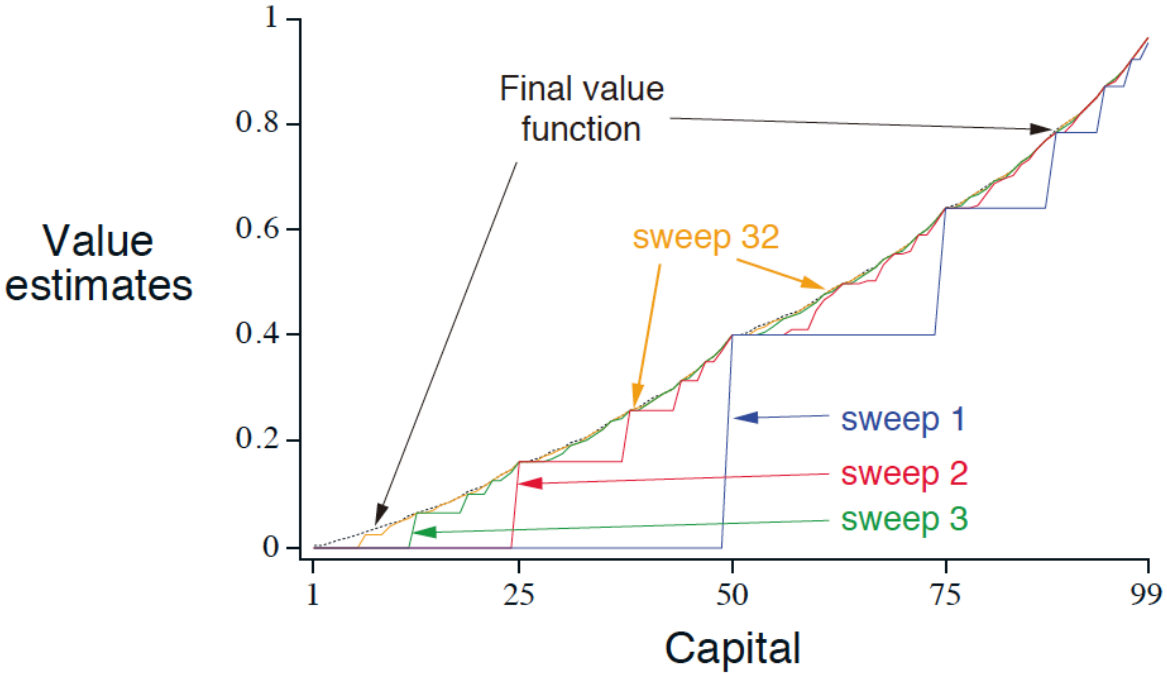
Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $\quad v \leftarrow V(s)$
| $\quad V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$
| $\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

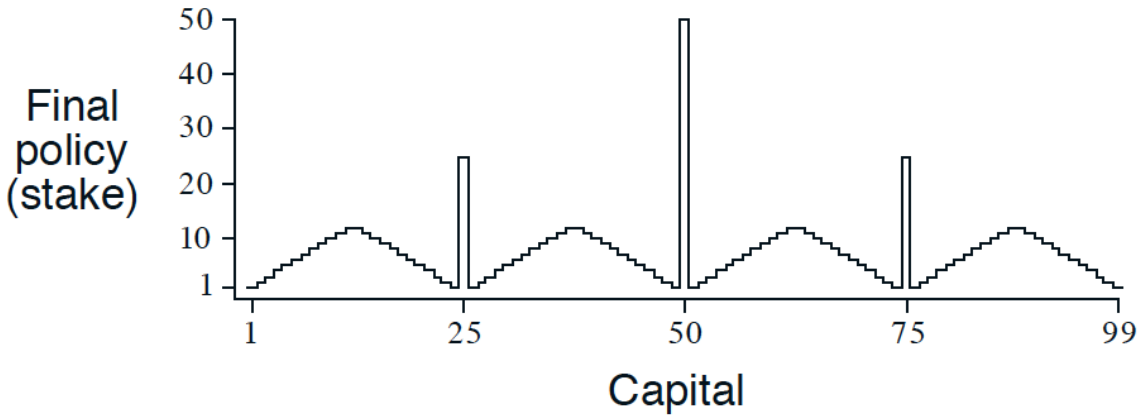
Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$

Example: Gambler's Problem

- Gambler can repeatedly bet \$ on a coin flip
- Heads he wins his stake, tails he loses it
- Initial capital $\in \{\$1, \$2, \ldots, \$99\}$
- Gambler wins if his capital becomes \$100, loses if it become \$0
- Unfair coin: Heads (gambler wins) with probability $p = 0.4$
- States, Actions, Rewards? Discounting?

Solution


Value estimates
1
0.8
0.6
0.4
0.2
0
Final value function
sweep 32
sweep 1
sweep 2
sweep 3
1
25
50
75
99
Capital

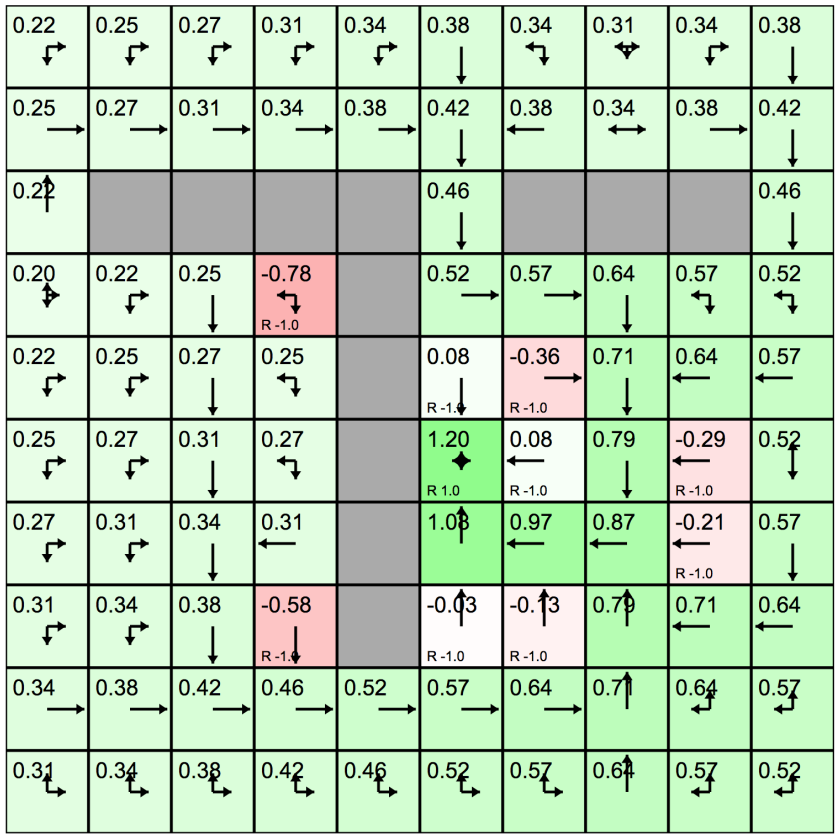

Final policy (stake)
50
40
30
20
10
1
1
25
50
75
99
Capital

Example of Value Iteration in Practice

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

Demo using REINFORCEjs Library

- Demo link: `https://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_dp.html`
- Gridworld DP demo: policy evaluation, policy improvement, value iteration



Policy Iteration vs. Value Iteration

- Policy iteration
  - picks a policy and then determines the true, steady-state value of being in each state given the policy.
  - Given this value, a new policy is chosen.
  - Converges faster in terms of the number of iterations, since it doing a lot more work in each iteration
- Value iteration
  - updates the value at each iteration
  - and then determines a new policy given the new estimate of the value function.
  - At any iteration, the value function is not the true, steady-state value of the policy
  - Much faster per iteration, may be far from the true value function

Policy Iteration vs. Value Iteration

- Policy iteration includes: policy evaluation + policy improvement, and the two are repeated iteratively until policy converges.
- Value iteration includes: finding optimal value function + one policy extraction. There is no repeat of the two because once the value function is optimal, then the policy out of it should also be optimal (i.e. converged).
- Finding optimal value function can also be seen as a combination of policy improvement (due to max) and truncated policy evaluation (the reassignment of v(s) after just one sweep of all states regardless of convergence).

Outline

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

Efficiency of Dynamic Programming

- To find an optimal policy is polynomial in the number of states
- BUT, the number of states is often astronomical, e.g., often growing exponentially with the number of state variables (what Bellman called "the curse of dimensionality").
- In practice, classical DP can be applied to problems with a few millions of states

Asynchronous Dynamic Programming

- DP methods described so far used *synchronous* backups
- i.e. all states are backed up in parallel
- require exhaustive sweeps of the entire state set
- *Asynchronous DP* backs up states individually, in any order
- Sample a state according to some rule, then apply the appropriate backup
- Can significantly reduce computation (does not get locked into hopelessly long sweeps)
- Guaranteed to converge if all states continue to be selected

Asynchronous Dynamic Programming

Three simple ideas for asynchronous dynamic programming:

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

In-Place Dynamic Programming

- Synchronous value iteration stores two copies of value function

  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 because the values are updated **in place**

  for all s in $\mathcal{S}$

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

Prioritized Sweeping

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

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

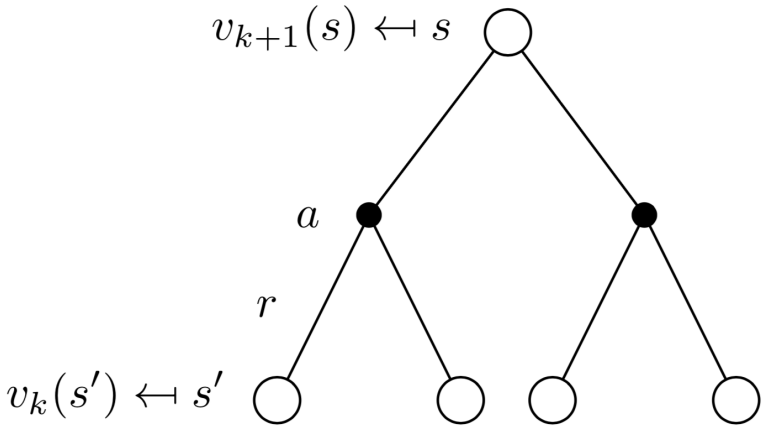
- Backup the state with the largest remaining Bellman error
- Update Bellman error of affected states after each backup
- Requires knowledge of reverse dynamics (predecessor states)
- Can be implemented efficiently by maintaining a priority queue

Real-Time Dynamic Programming

- 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

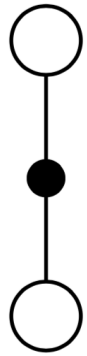
$$v(S_t) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}^a_{S_t} + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{S_t s'} v(s') \right)$$

Full-Width Backups

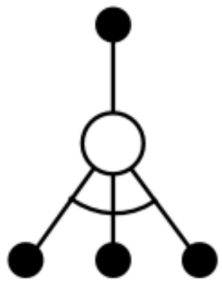


- DP uses *full-width* backups
- For each backup (sync or async)
  - Every successor state and action is considered
  - Using knowledge of the MDP transitions and reward function
- DP is effective for medium-sized problems (millions of states)
- For large problems DP suffers Bellman's *curse of dimensionality*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

Sample Backups



- The key design for RL algorithms such as Q-learning and SARSA in subsequent lectures
- Using sample rewards and sample transitions $\langle S, A, R, S' \rangle$
- Instead of reward function $\mathcal{R}$ and transition dynamics $\mathcal{P}$
- Advantages:
  - Model-free: no advance knowledge of MDP required
  - Breaks the curse of dimensionality through sampling
  - Cost of backup is constant, independent of $n = |S|$



Outline

Approximate Dynamic Programming

- Find approximately optimal policies for problems with large or continuous spaces
- Local and global approximation strategies for efficiently finding value functions and policies for known models.

Local Approximation

- Assumption: states close to each other have similar values
- Nearest neighbor: assign all weight to the closest discrete state, resulting in a piecewise constant value function
- k-nearest neighbor: a weight of $1/k$ is assigned to each of the k nearest discrete states of s.
- Kernel methods
- Linear interpolation
- Simplex-based interpolation

Global Approximation

- Uses a fixed set of parameters $\{\lambda_i\}$ to approximate the value function over the entire state space
- Linear Regression

Algorithm 4.5 Linear regression value iteration

```
function LinearRegressionValueIteration
    λ ← 0
    loop
        for i ← 1 to n
            u_i ← max_a [R(s_i, a) + γ Σ_{s'} T(s' | s_i, a) λ^⊤ β(s')]
        λ_{1:m} ← Regress(β, s_{1:n}, u_{1:n})
    return λ
```

Nonlinear Function Approximation

- 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 $\{\langle s, \tilde{v}_k(s) \rangle\}$
- Key idea behind the Deep Q-Learning

Online Methods

- Restrict computation to states that are reachable from the current state
- Such reachable state space can be orders of magnitude smaller than the full state space
- Significantly reduce the amount of storage and computation required to choose optimal (or approximately optimal) actions

Online Methods

- Forward search: online action-selection method that looks ahead from some initial state s to some horizon (or depth) d.
- Branch and bound search: uses knowledge of the upper and lower bounds of the value function to prune portions of the search tree
- Sparse sampling
  - Avoid the worst-case exponential complexity of forward and branch-and-bound search
  - Uses a generative model to produce samples of the next state and reward
  - The run time complexity is still exponential in the horizon but does not depend on the size of the state space

Monte Carlo Tree Search

- The complexity of Monte Carlo tree search does not grow exponentially with the horizon
- Upper Confidence Bound for Trees (UCT) implementation

Algorithm 4.9 Monte Carlo tree search

```
function SelectAction(s, d)
    loop
        Simulate(s, d, π_0)
    return arg max_a Q(s, a)
function Simulate(s, d, π_0)
    if d = 0
        return 0
    if s ∉ T
        for a ∈ A(s)
            (N(s,a), Q(s,a)) ← (N_0(s,a), Q_0(s,a))
        T = T ∪ {s}
        return Rollout(s, d, π_0)
    a ← arg max_a Q(s,a) + c sqrt(log N(s) / N(s,a))
    (s', r) ~ G(s,a)
    q ← r + γ Simulate(s', d − 1, π_0)
    N(s,a) ← N(s,a) + 1
    Q(s,a) ← Q(s,a) + (q − Q(s,a)) / N(s,a)
    return q
```

Direct Policy Search

- So far we presented methods that involve computing or approximating the value function
- An alternative is to search the space of policies directly.
  - Although the state space may be high dimensional
  - making approximation of the value function difficult
  - the space of possible policies may be relatively low dimensional and can be easier to search directly.
- Example: local search methods
  - also known as hill climbing or gradient ascent
  - begins at a single point in the search space and then incrementally moves from neighbor to neighbor in the search space until convergence
- Other examples: Markov approximation & simulated annealing & evolutionary method (genetic algorithm)

Cross Entropy Method for Policy Search

- maintain a distribution over policies and updates the distribution based on policies that perform well
- Consists of two stages
  - Sample: draw n samples from $P(\lambda; \theta)$, evaluate the performance and then sort samples according to a decreasing order of performance.
  - Update: use the top m samples (called "elite samples") to update θ using cross entropy minimization, which is the MLE based on the m top-performing samples.

Cross Entropy Algorithms

Algorithm 4.12 Cross entropy policy search

```
function CrossEntropyPolicySearch(θ)
    repeat
        for i ← 1 to n
            λ_i ~ P(·; θ)
            v_i ← MonteCarloPolicyEvaluation(λ_i)
        Sort (λ_1, ... λ_n) in decreasing order of v_i
        θ ← arg max_θ Σ_{j=1}^{m} log P(λ_j | θ)
    until convergence
    return λ ← arg max P(λ | θ)
```

Cross Entropy Algorithms: Example

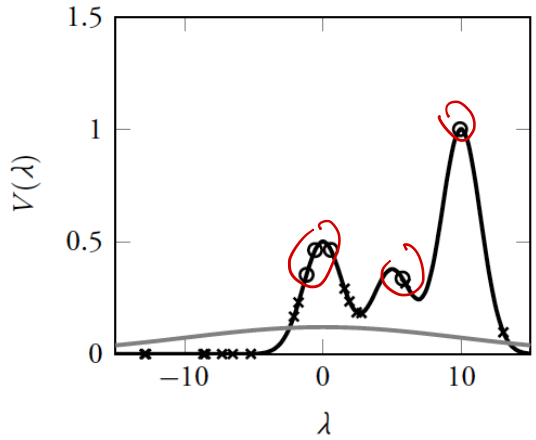
$\theta = (\mu, \sigma)$

$p(\lambda|\theta) \sim N(\lambda|\mu, \sigma^2)$

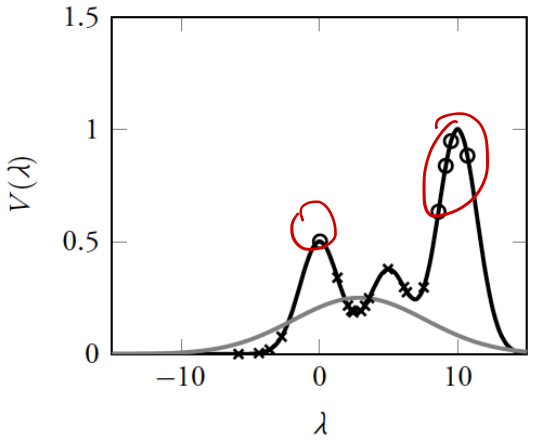
Initially $\theta = (0, 10)$

$n = 20$ samples

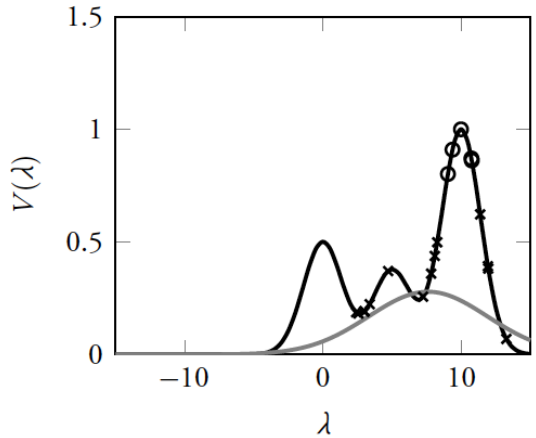
$h = 5$ elite samples (shown with circles)

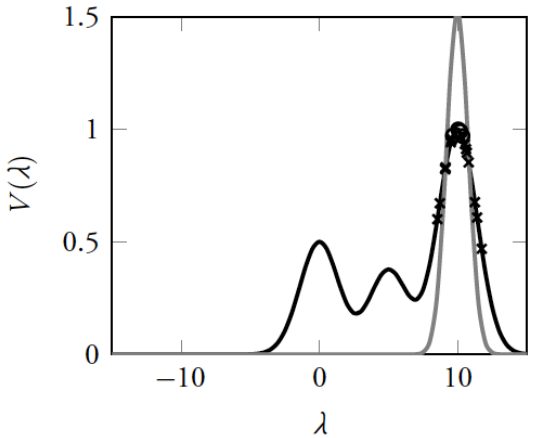


(a) Iteration 1.



(b) Iteration 2.





Find the global optimum.

Outline

Iterative Algorithms

- $x_i(t)$: i^{th} component of $x(t)$
- $f_i(t)$: i^{th} component of function $f(t)$
- Jacob-type iteration (components are simultaneously updated): for $i = 1, \ldots, n$

$$x_i(t+1) = f_i(x_1(t), \ldots, x_n(t))$$

- Gauss-Seidel iteration (updated one component at a time, most recently updated values are used): for $i = 1, \ldots, n$

$$x_i(t+1) = f_i(x_1(t+1), \ldots, x_{i-1}(t+1), x_i(t), \ldots, x_n(t))$$

most recent update

Iterative Algorithms

- Gauss-Seidel algorithms are often preferable
  - incorporate the newest available information
  - sometime converge faster than the corresponding Jacobi-type algorithm
- A single Gauss-Seidel iteration is called a "sweep"
- One example: Gibbs Sampling with systematic scan

Gauss-Seidel Variation of Value Iteration

- Also called "In-place value iteration"
- We have to loop over all states with an order

Step 1′. For each $s \in \mathcal{S}$ compute

$$v^n(s) = \max_{a \in \mathcal{A}} \left\{ C(s,a) + \gamma \left(\sum_{s' < s} \mathbb{P}(s'|s,a) v^n(s') + \sum_{s' \geq s} \mathbb{P}(s'|s,a) v^{n-1}(s') \right) \right\}$$

order

Outline

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

Norm

Definition (Norm)

A nonnegative real-valued function $\|\cdot\|$ defined on a vector space is called a norm if

1. $\|x\| = 0 \Leftrightarrow x = 0$
2. $\|x + y\| \leq \|x\| + \|y\|$ (triangle inequality)
3. $\|\alpha x\| = |\alpha|\|x\|$

∞-Norm for Value Functions

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

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

- We also have

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

∞-Norm for Matrix

- We define ∞-norm of a matrix P as follows:

$$\|P\|_\infty = \max_{s \in \mathcal{S}} \sum_{j \in \mathcal{S}} |P(s,j)|$$

- That is the largest row sum of the matrix.
- If P is a one-step transition matrix (stochastic matrix), then $\|P\|_\infty = 1$

Banach Space

- The vector space $\mathcal{V}$ over value functions is a normed linear space
  - it is closed under addition and scalar multiplication
  - and it has a norm
- A sequence $v_n \in \mathcal{V}$, $n = 1, 2, \ldots$, is said to be a Cauchy sequence if for all $\epsilon > 0$, there exists N such that for all $n, m \geq N$: $\|v_n - v_m\| < \epsilon$.
- A normed linear space is complete if every Cauchy sequence contains a limit point in that space.
- A Banach space is a complete normed linear space.

Contraction Mapping

Definition (Contraction Mapping)

$T : \mathcal{V} \to \mathcal{V}$ is a contraction mapping if there exists a γ, $0 \leq \gamma < 1$, such that

$$\|Tv - Tu\| \leq \gamma \|v - u\|$$

Banach Fixed-Point Theorem

Theorem (Banach Fixed-Point Theorem)

Let $\mathcal{V}$ be a Banach space, and let $T : \mathcal{V} \to \mathcal{V}$ be a contraction mapping. Then

1. *there exists a unique $v^* \in \mathcal{V}$ such that $Tv^* = v^*$*
2. *for arbitrary $v_0 \in \mathcal{V}$, the sequence v_n defined by $v_{n+1} = Tv_n = T^{n+1}v_0$ converges to v^**

v^* is a fixed point.

Contraction Mapping Theorem

Theorem (Contraction Mapping Theorem)

For any Banach space $\mathcal{V}$ with a γ-contraction T,

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

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

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

$\|P^\pi\|_\infty = 1$

Convergence of Policy Evaluation

- 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

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)

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

Another Proof

① Let U and V be two value functions in value space $\mathbb{V}$ and any state $s \in \mathcal{S}$.

W.L.O.G. we assume $T^*V(s) \geq T^*U(s)$

Considering a_s^* such that $a_s^* \in \underset{a \in \mathcal{A}}{\operatorname{argmax}} \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V(s') \right]$

Then we can write

$$0 \leq |T^*V(s) - T^*U(s)| = T^*V(s) - T^*U(s)$$

Since $T^*V(s) = r(s, a_s^*) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a_s^*) V(s')$

$$T^*U(s) = \max_{a \in \mathcal{A}} \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) U(s') \right]$$

$$\geq r(s, a_s^*) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a_s^*) U(s')$$

$$\Rightarrow 0 \leq T^*V(s) - T^*U(s) \leq r(s, a_s^*) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a_s^*) V(s') - \left[r(s, a_s^*) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a_s^*) U(s') \right]$$

$$= \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a_s^*) (V(s') - U(s')) \leq \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a_s^*) \|V - U\|_\infty \quad \left(\sum_{s' \in \mathcal{S}} p(s'|s, a_s^*) = 1 \right)$$

$$= \gamma \|V - U\|_\infty$$

Another Proof

② if $T^*v(s) < T^*u(s)$, with the same method, we have

$$0 \le T^*u(s) - T^*v(s) \le \gamma \|v-u\|_\infty$$

Thus we have $\forall s \in S$,

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

Consequently,

$$\|T^*(v) - T^*(u)\|_\infty = \max_{s \in S} |T^*v(s) - T^*u(s)| \le \gamma \|v-u\|_\infty$$

$\Rightarrow T^*$ is a γ-contraction.

Q.E.D.

Another Proof

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_*

Policy Improvement Theorem

Theorem (Policy Improvement Theorem)

Let $\mathcal{D}$ denotes the set of stationary, Markov and deterministic policy. For any policy π^+ defined by

$$\pi^+ \in \arg\max_{\delta \in \mathcal{D}} \{r_\delta + \gamma P_\delta V^\pi\},$$

One-step improvement
greedy improvement.

we have $V^{\pi^+} \geq V^\pi$. The equality holds iff $\pi = \pi^*$

Proof

Since $\pi^\dagger \in \underset{a \in \mathcal{D}}{\operatorname{argmax}} \{ r_a + \gamma P_a V^\pi \}$

$$\Rightarrow \quad r_{\pi^\dagger} + \gamma P_{\pi^\dagger} V^\pi = \max_{a \in \mathcal{D}} \{ r_a + \gamma P_a V^\pi \}$$

$$\geq r_\pi + \gamma P_\pi V^\pi = V^\pi \quad \text{(Bellman Expectation Equation under } \pi\text{)}$$

thus $r_{\pi^\dagger} + \gamma P_{\pi^\dagger} V^\pi \geq V^\pi$

$$\Rightarrow \quad r_{\pi^\dagger} + \gamma P_{\pi^\dagger} V^{\pi^\dagger} + \gamma P_{\pi^\dagger} (V^\pi - V^{\pi^\dagger}) \geq V^\pi$$

Bellman Expectation equation under $\pi^\dagger$ $\Rightarrow$

$$V^{\pi^\dagger} + \gamma P_{\pi^\dagger} (V^\pi - V^{\pi^\dagger}) \geq V^\pi$$

$$\Rightarrow \quad V^{\pi^\dagger} - \gamma P_{\pi^\dagger} V^{\pi^\dagger} \geq V^\pi - \gamma P_{\pi^\dagger} V^\pi$$

$$\Rightarrow \quad (I - \gamma P_{\pi^\dagger}) V^{\pi^\dagger} \geq (I - \gamma P_{\pi^\dagger}) V^\pi$$

$$\Rightarrow \quad V^{\pi^\dagger} \geq V^\pi$$

Proof

On the other hand, if $u \geq v$, $(I - \gamma P_{\pi^+})^{-1} u = u + \gamma P_{\pi^+} u + \gamma^2 P_{\pi^+}^2 u + \cdots$

$$\geq v + \gamma P_{\pi^+} v + \gamma^2 P_{\pi^+}^2 v^2 + \cdots$$

$$= (I - \gamma P_{\pi^+})^{-1} v.$$

Thus equality hold iff

$$\max_{s \in \mathcal{O}} \{ r_s + \gamma P_s v^\pi \} = v^\pi \quad \text{(Bellman optimality equation)}$$

$$\Rightarrow v^\pi = v^{\pi^*}$$

$$\Rightarrow \pi = \pi^*$$

Proof

Convergence of Policy Iteration

- By policy improvement theorem we know that
- the sequence of V_n is a monotonic, increasing sequence of value functions.
- It is upper bounded by v_* and converges.
- Thus policy iteration converges on v_* with optimal policy π^*

Outline

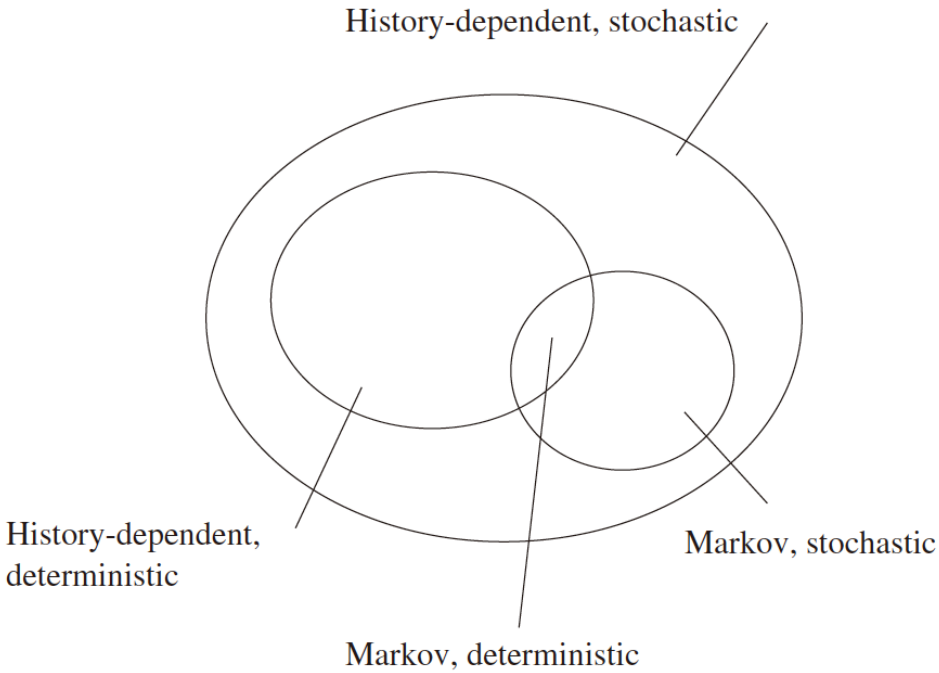
Markov Models in General

Markov Models		Do we have control over the state transitions?	
		No	Yes
Are the states completely observable?	Yes	Markov Chain	MDP Markov Decision Process
	No	HMM Hidden Markov Model	POMDP Partially Observable Markov Decision Process

Different Policy Families for MDP

Policy π_t	Deterministic	Stochastic
Markov	$s_t \longrightarrow a_t$	$a_t,\ s_t \longrightarrow [0, 1]$
History-dependent	$h_t \longrightarrow a_t$	$h_t,\ s_t \longrightarrow [0, 1]$

Different Policy Families for MDP



Main Performance Criteria of MDP

- The finite criterion

$$\mathbb{E}\left[r_0 + r_1 + \cdots + r_{N-1} \middle| s_0\right]$$

- The discounted criterion

$$\mathbb{E}\left[r_0 + \gamma r_1 + \gamma^2 r_2 \cdots + \gamma^t r_t + \cdots \middle| s_0\right]$$

- The total reward criterion

$$\mathbb{E}\left[r_0 + r_1 + \cdots + r_t + \cdots \middle| s_0\right]$$

- The average criterion

$$\lim_{n\to\infty} \frac{1}{n} \mathbb{E}\left[r_0 + r_1 + \cdots + r_{n-1} \middle| s_0\right]$$

Dynamic Programming Algorithms

- Policy evaluation: backups without a max
- Policy improvement: form a greedy policy, if only locally
- Policy iteration: alternate the above two processes
- Value iteration: backups with a max
- Full backups (to be contrasted later with sample backups)
- Generalized Policy Iteration (GPI)
- Asynchronous DP: a way to avoid exhaustive sweeps
- Bootstrapping: updating estimates based on other estimates
- Biggest limitation of DP is that it requires a probability model (as opposed to a generative or simulation model)

Curse of modeling

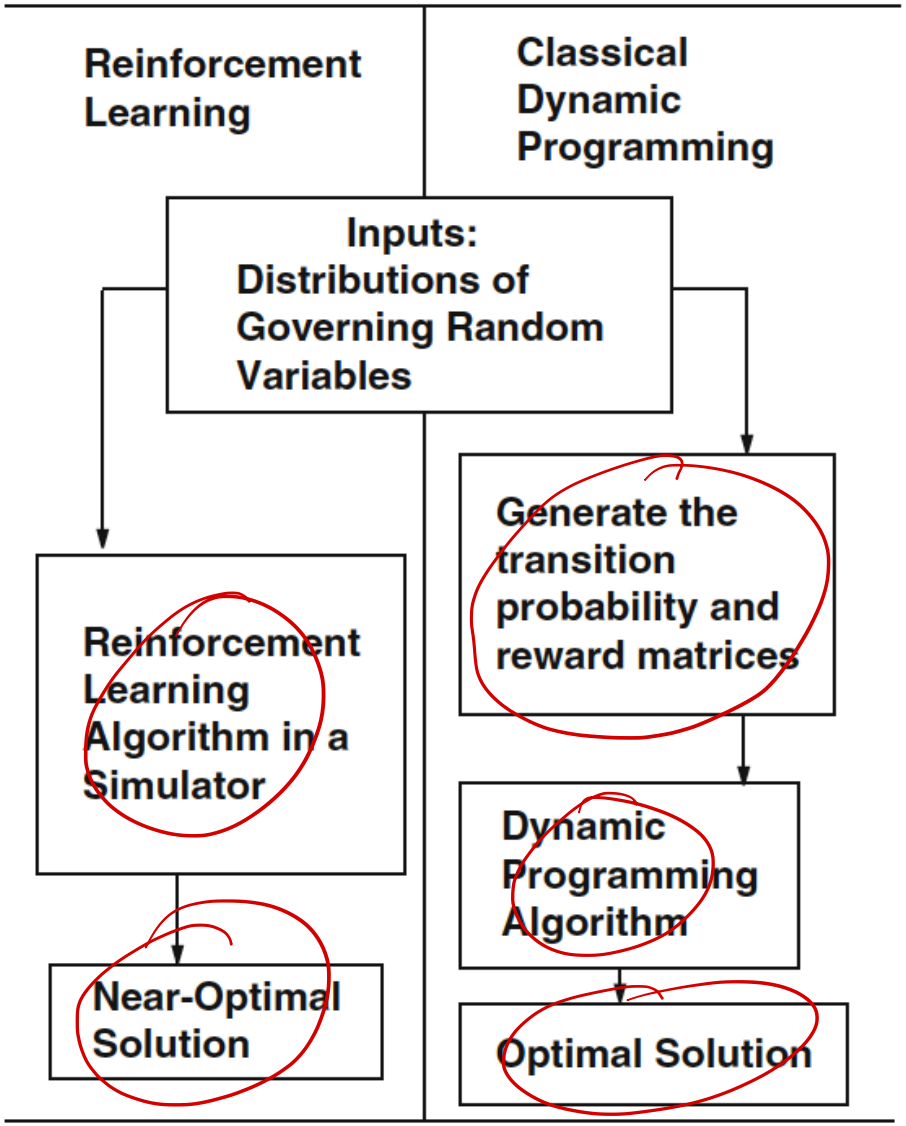
Dynamic Programming Algorithms

Problem	Bellman Equation	Algorithm
Prediction	Bellman Expectation Equation	Iterative Policy Evaluation
Control	Bellman Expectation Equation	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

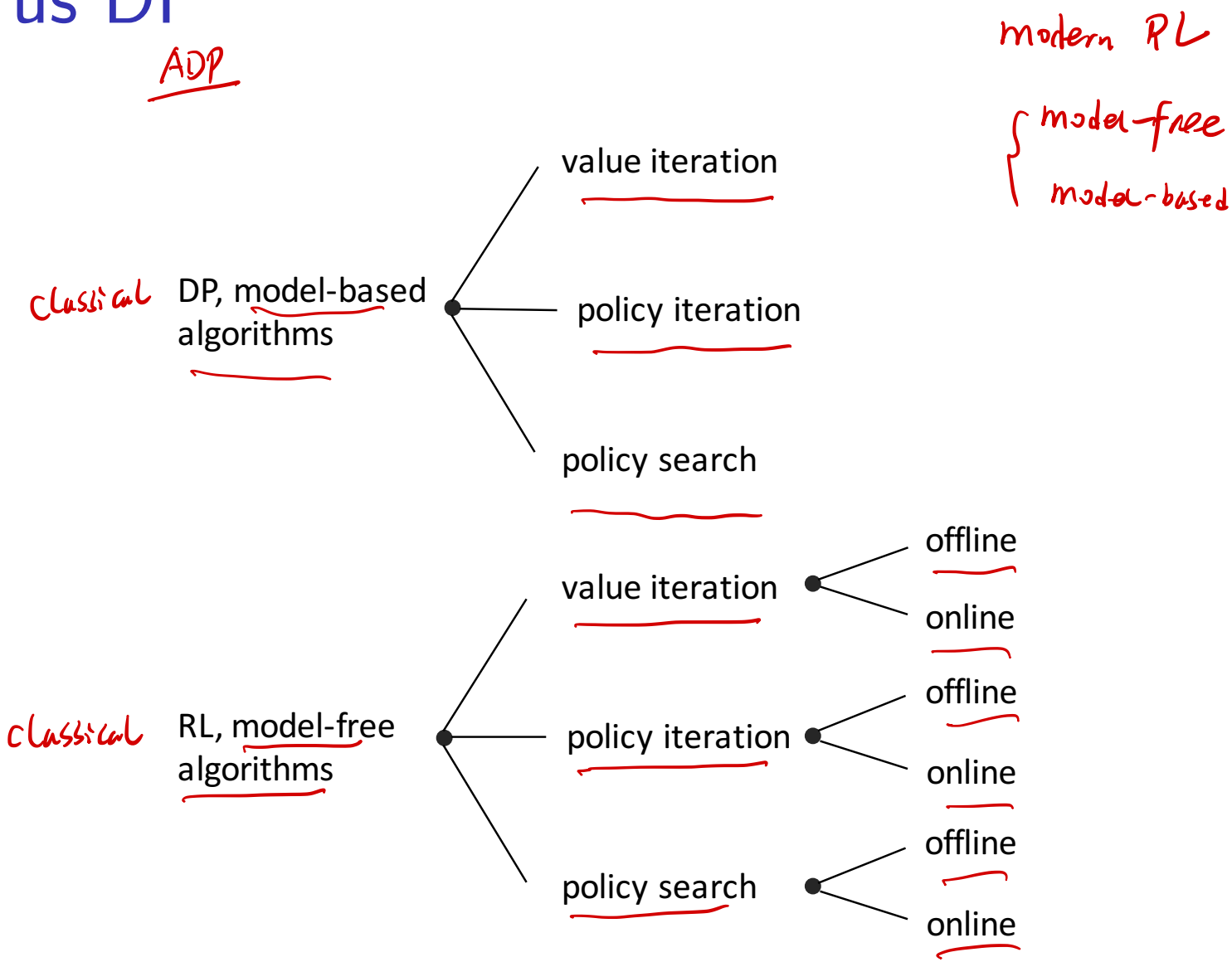
A Comparison of DP & RL & Heuristics

Method	Level of modeling effort	Solution quality
DP	High	High
RL	Medium	High
Heuristics	Low	Low

Difference Between DP & RL



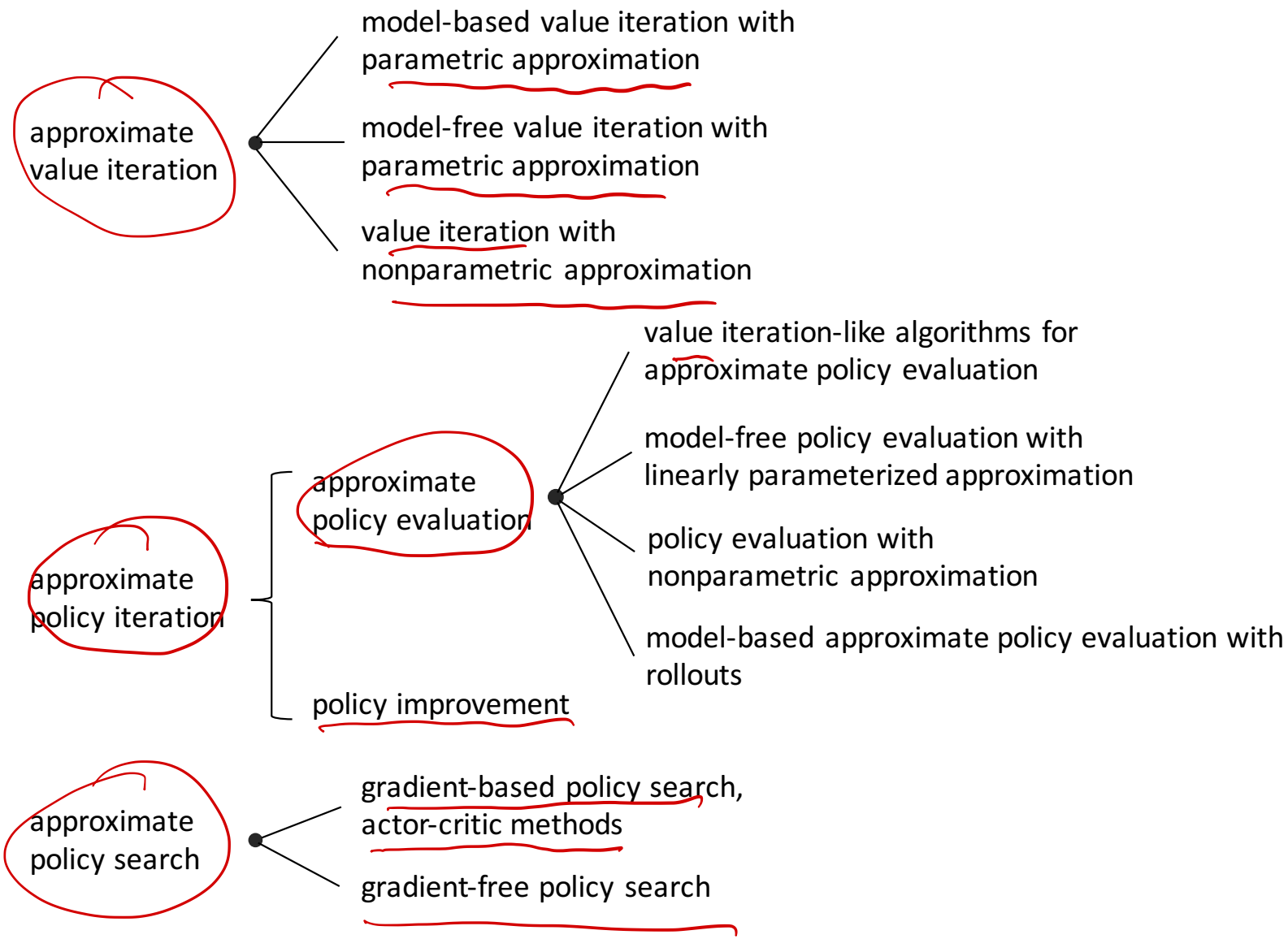
RL verus DP


ADP
modern RL
model-free
model-based
classical
DP, model-based algorithms
value iteration
policy iteration
policy search
classical
RL, model-free algorithms
value iteration
offline
online
policy iteration
offline
online
policy search
offline
online

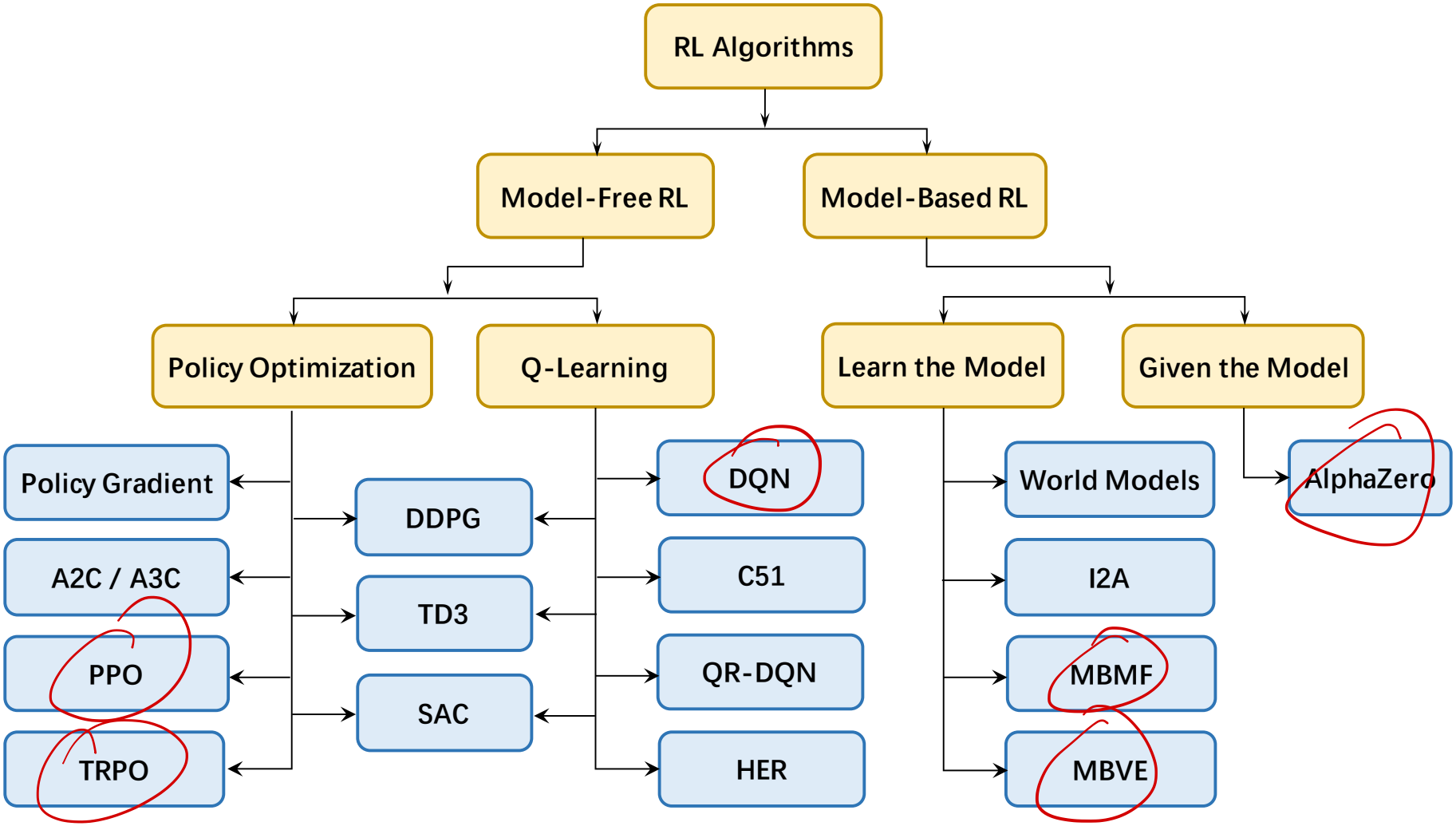
Methods for Exploration & Exploitation Trade-Off

- Undirected methods: use few information from the learning experiments beyond the value function itself
  - ϵ-greedy
  - Boltzmann exploration (softmax)
- Directed methods: use specific exploration heuristics based on the information available from learning
  - adding some exploration bonus to $Q(s, a)$
  - e.g., bonus is related to the number of times action a was chosen in state s.

Approximate RL Methods



RL Algorithms: State-of-the-Art



Outline

Main References

- Reinforcement Learning: An Introduction (second edition), R. Sutton & A. Barto, 2018.
- Markov Decision Processes: Discrete Stochastic Dynamic Programming, (second edition), Martin L. Puterman, 2005.
- RL course slides from Richard Sutton, University of Alberta.
- RL course slides from David Silver, University College London.