

Lecture 8: Dynamic Programming

Ziyu Shao

School of Information Science and Technology
ShanghaiTech University

May 17 & 19, 2021

Outline

- 1 Introduction
- 2 Policy Evaluation
- 3 Policy Iteration
- 4 Value Iteration
- 5 Extensions to Dynamic Programming
- 6 Approximate Dynamic Programming
- 7 Iterative Algorithms
- 8 Summary & Lookahead
- 9 References

Outline

- 1 Introduction
- 2 Policy Evaluation
- 3 Policy Iteration
- 4 Value Iteration
- 5 Extensions to Dynamic Programming
- 6 Approximate Dynamic Programming
- 7 Iterative Algorithms
- 8 Summary & Lookahead
- 9 References

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

	state values	action values
prediction	V_{π}	q_{π}
control	V_{*}	q_{*}

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 $\pi(a|s) : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ where $\pi(a|s)$ denotes the probability that action a may be chosen in state s

Outline


- 1 Introduction
- 2 Policy Evaluation
- 3 Policy Iteration
- 4 Value Iteration
- 5 Extensions to Dynamic Programming
- 6 Approximate Dynamic Programming
- 7 Iterative Algorithms
- 8 Summary & Lookahead
- 9 References

Iterative Policy Evaluation

- Problem: evaluate a given policy π
- Solution: iterative application of Bellman expectation backup
- $v_1 \rightarrow v_2 \dots \rightarrow v_\pi$
- 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

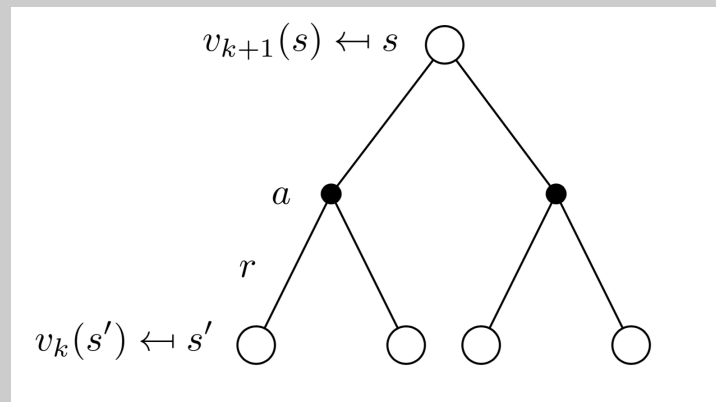
Iterative Policy Evaluation: Sweep

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

a “sweep” 

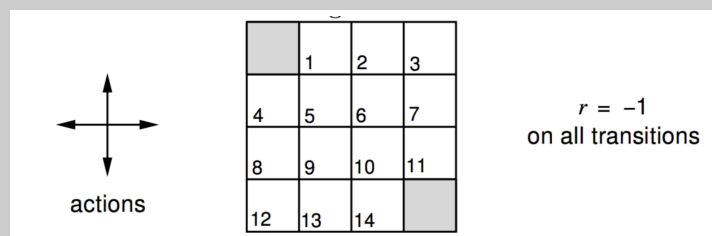
A sweep consists of applying a **backup operation** to each state.

Iterative Policy Evaluation: Backup



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

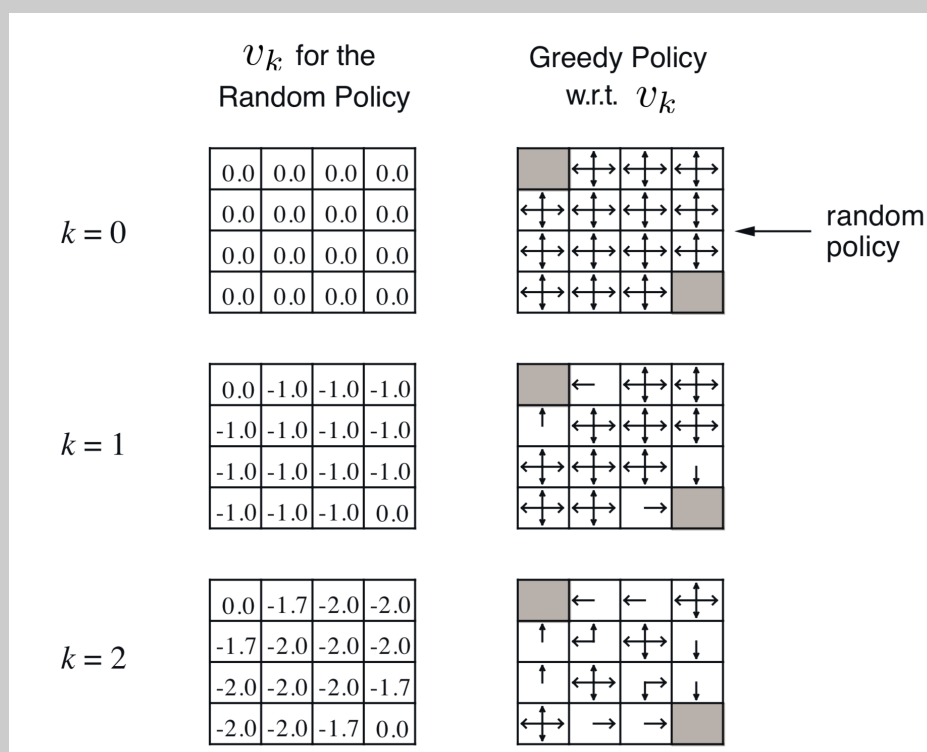
Example: Small Gridworld



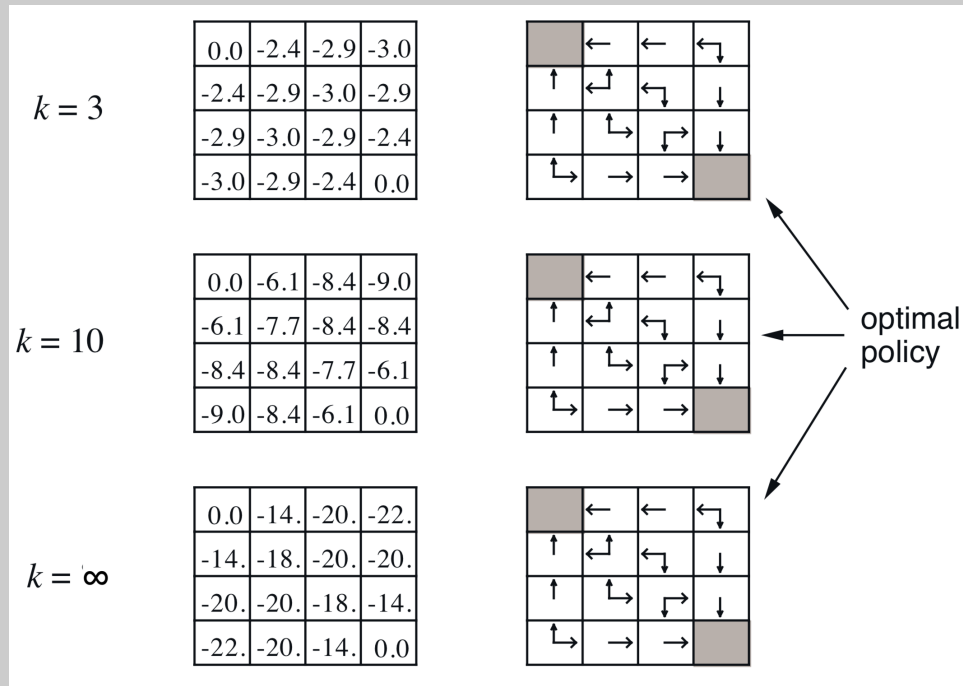
- Undiscounted episodic MDP ($\gamma = 1$)
- Nonterminal states 1, ..., 14
- One terminal state (shown twice as shaded squares)
- Actions leading out of the grid leave state unchanged
- Reward is -1 until the terminal state is reached
- Agent follows uniform random policy

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

Iterative Policy Evaluation in Small Gridworld



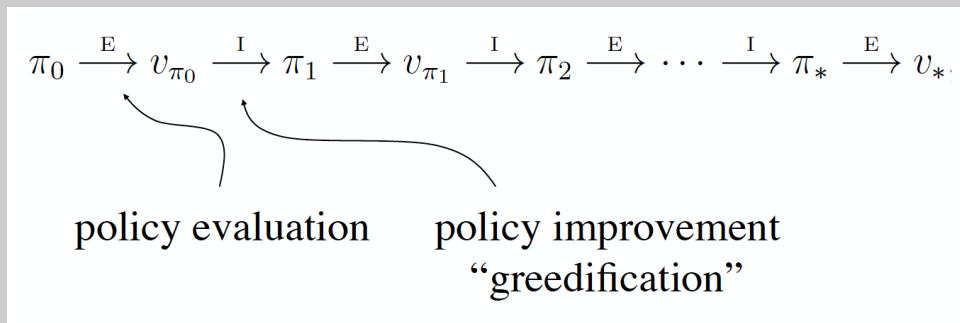
Iterative Policy Evaluation in Small Gridworld (2)



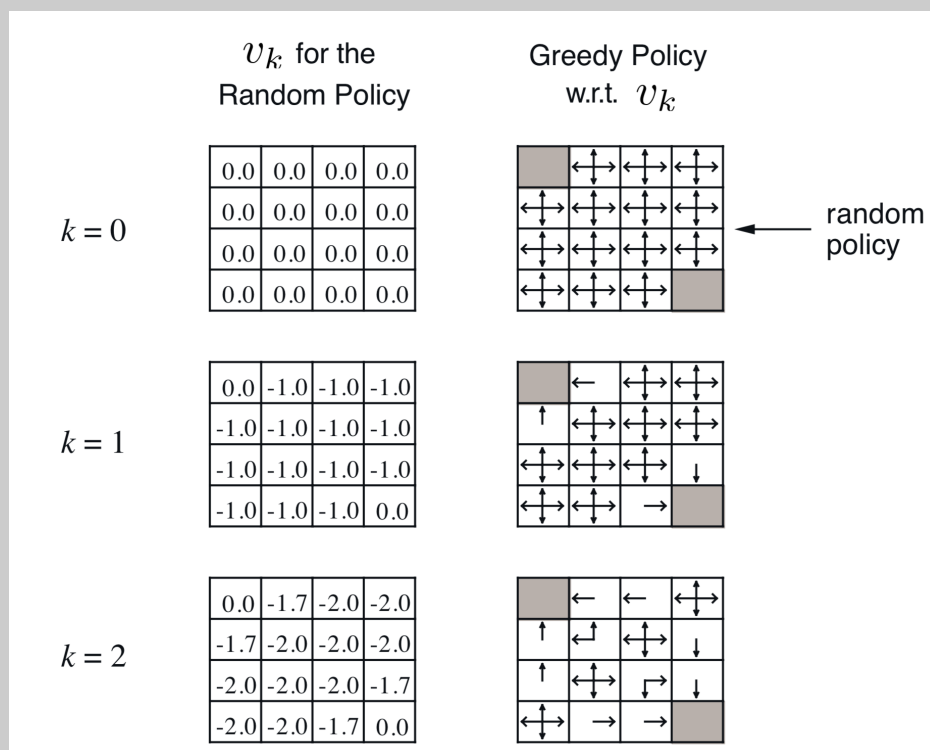
Outline

- 1 Introduction
- 2 Policy Evaluation
- 3 Policy Iteration**
- 4 Value Iteration
- 5 Extensions to Dynamic Programming
- 6 Approximate Dynamic Programming
- 7 Iterative Algorithms
- 8 Summary & Lookahead
- 9 References

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, $v_{\pi'}(s) \geq v_{\pi}(s)$

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

Remark & Proof

Remark & Proof

Remark & Proof

Policy Improvement (2)

- If improvements stop,

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

- Then the Bellman optimality equation has been satisfied

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

- 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

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

- Improve** the policy by acting greedily with respect to v_π

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

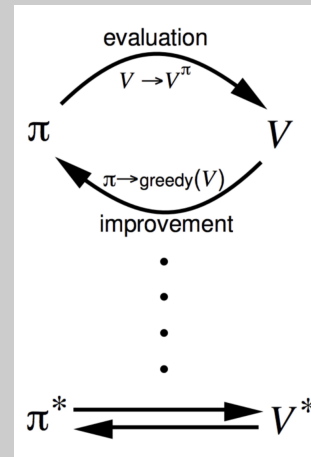
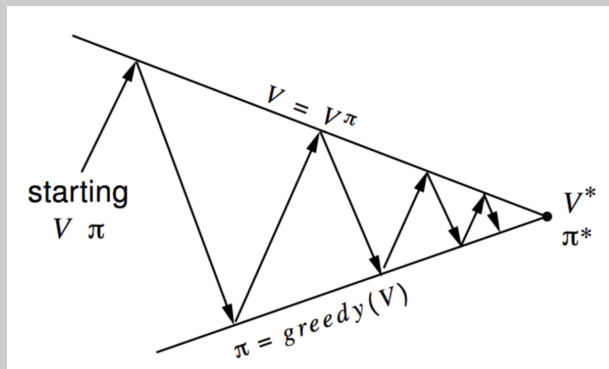
- In Small Gridworld improved policy was optimal, $\pi' = \pi^*$
- In general, need more iterations of improvement / evaluation
- But this process of **policy iteration** always converges to π^*

Policy Iteration Algorithm

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

- Initialization
 $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$
- Policy Evaluation
Loop:
 $\Delta \leftarrow 0$
 Loop for each $s \in \mathcal{S}$:
 $v \leftarrow V(s)$
 $V(s) \leftarrow \sum_{s',r} p(s', r | s, \pi(s)) [r + \gamma V(s')]$
 $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
 until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)
- Policy Improvement
 policy-stable \leftarrow true
 For each $s \in \mathcal{S}$:
 old-action $\leftarrow \pi(s)$
 $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r | s, a) [r + \gamma V(s')]$
 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 $\pi' \geq \pi$

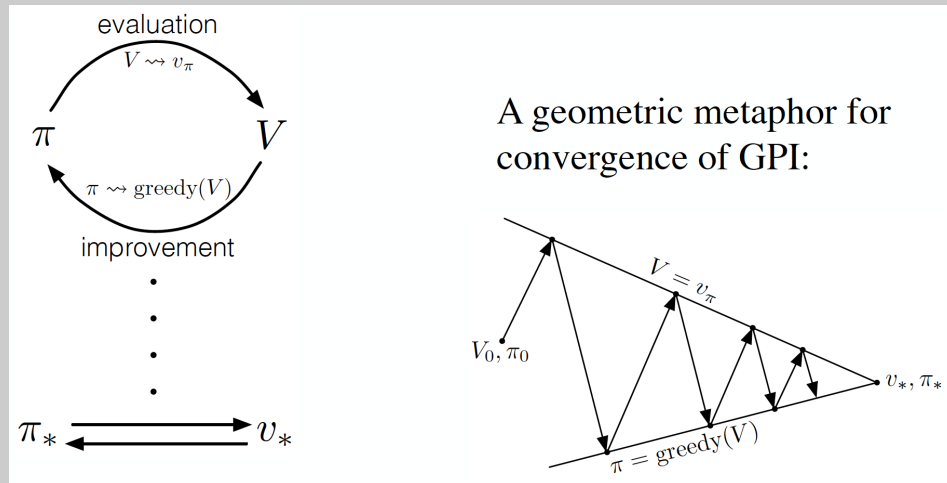
Greedy policy improvement

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)

- 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



Outline

- 1 Introduction
- 2 Policy Evaluation
- 3 Policy Iteration
- 4 Value Iteration**
- 5 Extensions to Dynamic Programming
- 6 Approximate Dynamic Programming
- 7 Iterative Algorithms
- 8 Summary & Lookahead
- 9 References

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

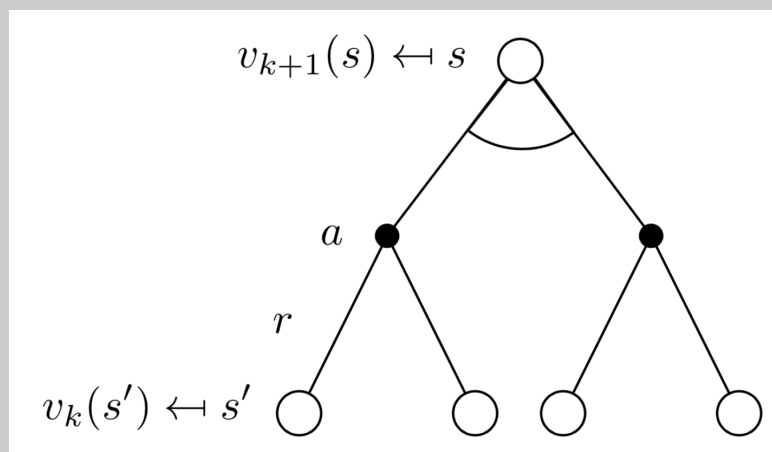
$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \left\{ \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right\}$$

- 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

Value Iteration

- Problem: find optimal policy π
- Solution: iterative application of Bellman optimality backup
- $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_*$
- 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

Value Iteration (2)



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

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 $V(\text{terminal}) = 0$

Loop:

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

Output a deterministic policy, $\pi \approx \pi_*$, such that
 $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$

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

- 1 Introduction
- 2 Policy Evaluation
- 3 Policy Iteration
- 4 Value Iteration
- 5 Extensions to Dynamic Programming**
- 6 Approximate Dynamic Programming
- 7 Iterative Algorithms
- 8 Summary & Lookahead
- 9 References

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 $O(m^2n^2)$ 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.

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

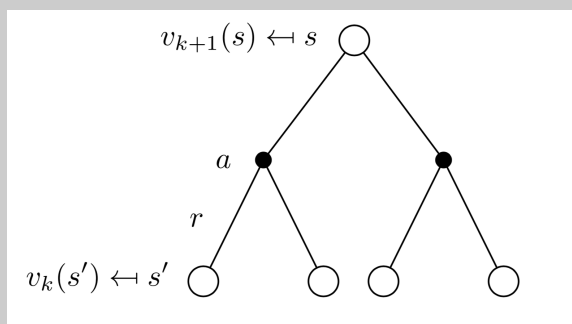
- Backup the state with the largest remaining Bellman error
- Update Bellman error of affected states after each backup
- Requires knowledge of reverse dynamics (predecessor states)
- Can be implemented efficiently by maintaining a priority queue

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

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

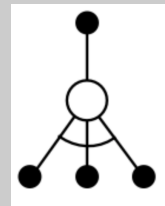
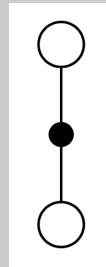
Full-Width Backups



- 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 $n = |\mathcal{S}|$ 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

- 1 Introduction
- 2 Policy Evaluation
- 3 Policy Iteration
- 4 Value Iteration
- 5 Extensions to Dynamic Programming
- 6 Approximate Dynamic Programming**
- 7 Iterative Algorithms
- 8 Summary & Lookahead
- 9 References

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

```
1: function LINEARREGRESSIONVALUEITERATION
2:    $\lambda \leftarrow \mathbf{0}$ 
3:   loop
4:     for  $i \leftarrow 1$  to  $n$ 
5:        $u_i \leftarrow \max_a [R(s_i, a) + \gamma \sum_{s'} T(s' | s_i, a) \lambda^\top \beta(s')]$ 
6:        $\lambda_{1:n} \leftarrow \text{REGRESS}(\beta, s_{1:n}, u_{1:n})$ 
7:   return  $\lambda$ 
```

Nonlinear Function Approximation

- Approximate the value function using a *function approximator* $\hat{v}(s, w)$
- Apply dynamic programming to $\hat{v}(\cdot, w)$
- 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,

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

- ▶ Train next value function $\hat{v}(\cdot, w_{k+1})$ 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

```
1: function SELECTACTION( $s, d$ )
2:   loop
3:     SIMULATE( $s, d, \pi_0$ )
4:   return  $\arg \max_a Q(s, a)$ 
5: function SIMULATE( $s, d, \pi_0$ )
6:   if  $d = 0$ 
7:     return 0
8:   if  $s \notin T$ 
9:     for  $a \in A(s)$ 
10:      ( $N(s, a), Q(s, a)$ )  $\leftarrow (N_0(s, a), Q_0(s, a))$ 
11:     $T = T \cup \{s\}$ 
12:    return ROLLOUT( $s, d, \pi_0$ )
13:    $a \leftarrow \arg \max_a Q(s, a) + c \sqrt{\frac{\log N(s)}{N(s, a)}}$ 
14:   ( $s', r$ )  $\sim G(s, a)$ 
15:    $q \leftarrow r + \gamma \text{SIMULATE}(s', d - 1, \pi_0)$ 
16:    $N(s, a) \leftarrow N(s, a) + 1$ 
17:    $Q(s, a) \leftarrow Q(s, a) + \frac{q - Q(s, a)}{N(s, a)}$ 
18:   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)

Outline

- 1 Introduction
- 2 Policy Evaluation
- 3 Policy Iteration
- 4 Value Iteration
- 5 Extensions to Dynamic Programming
- 6 Approximate Dynamic Programming
- 7 Iterative Algorithms**
- 8 Summary & Lookahead
- 9 References

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 $i = 1, \dots, n$

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

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

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

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

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

Outline

- 1 Introduction
- 2 Policy Evaluation
- 3 Policy Iteration
- 4 Value Iteration
- 5 Extensions to Dynamic Programming
- 6 Approximate Dynamic Programming
- 7 Iterative Algorithms
- 8 Summary & Lookahead**
- 9 References

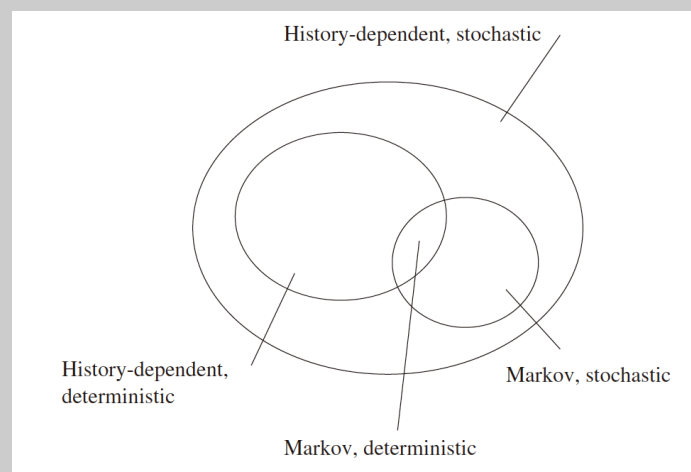
Markov Models in General

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

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

- The discounted criterion

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

- The total reward criterion

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

- The average criterion

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

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)

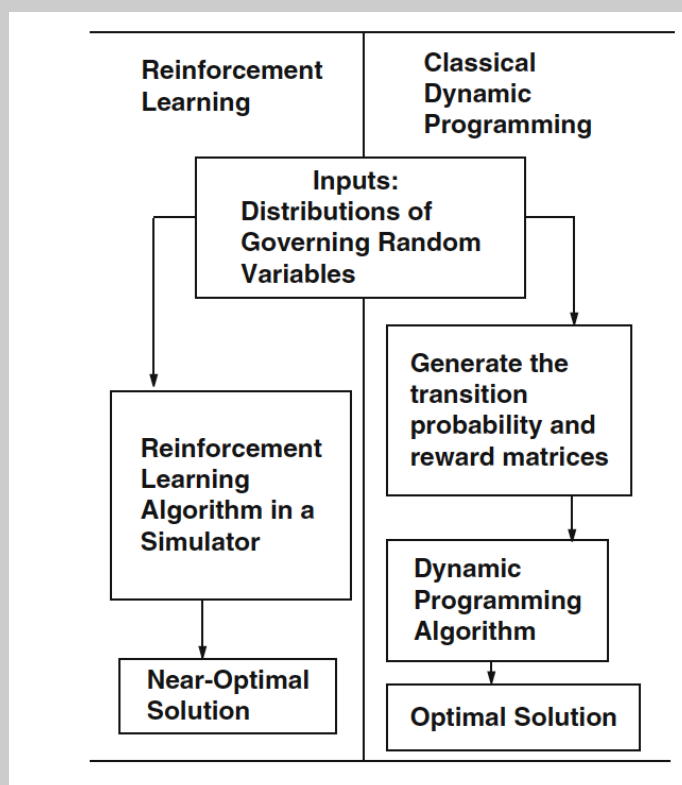
Dynamic Programming Algorithms

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

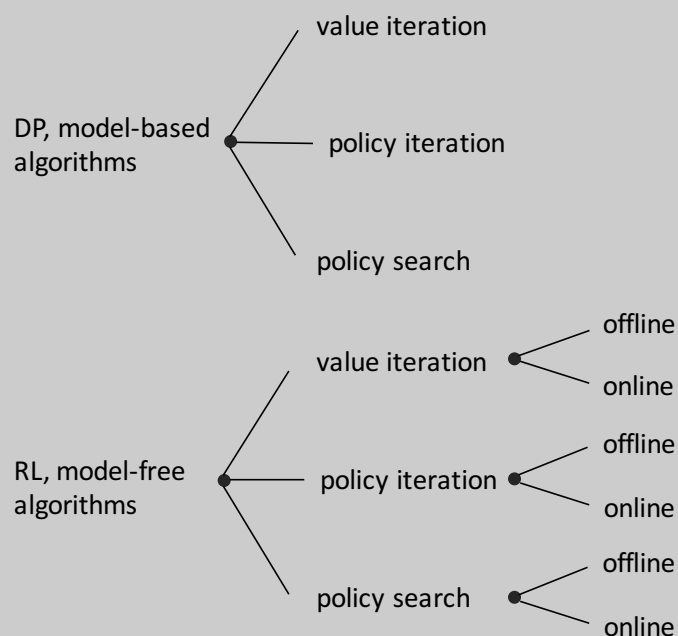
A Comparison of DP & RL & Heuristics

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

Difference Between DP & RL



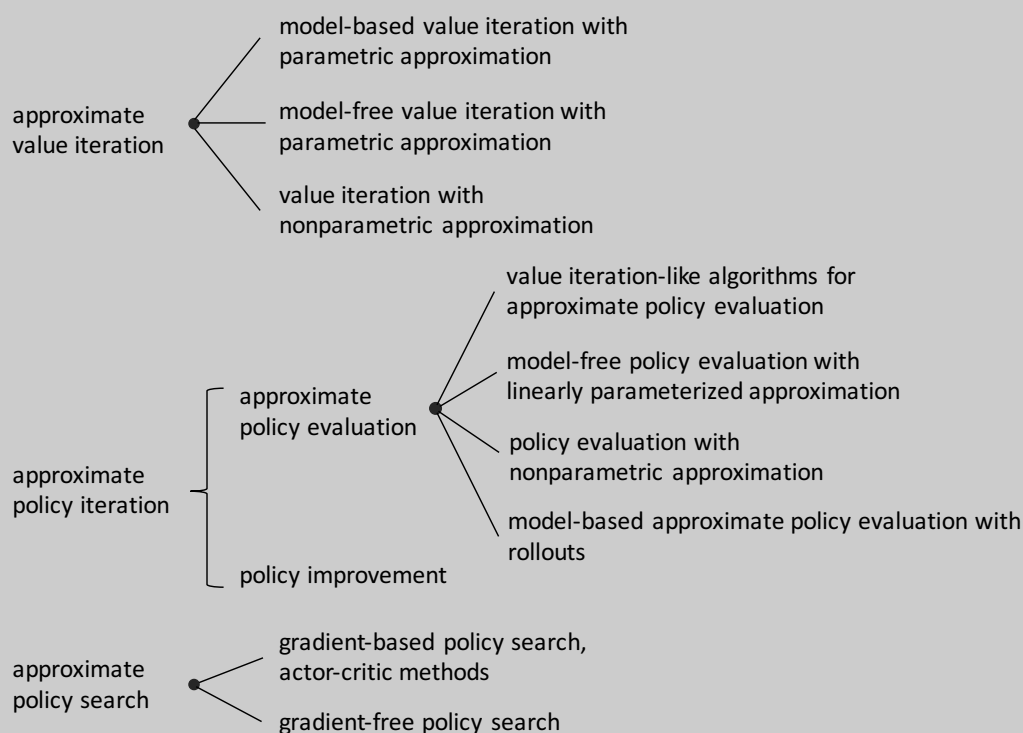
RL versus DP



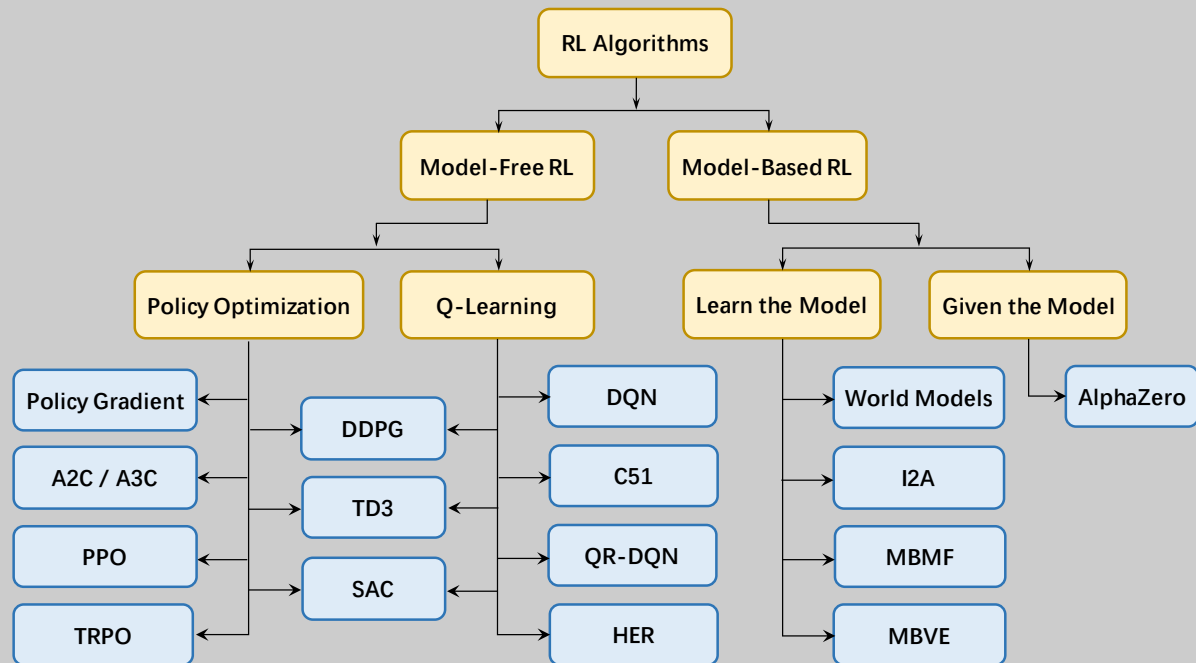
Methods for Exploration & Exploitation Trade-Off

- 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

- 1 Introduction
- 2 Policy Evaluation
- 3 Policy Iteration
- 4 Value Iteration
- 5 Extensions to Dynamic Programming
- 6 Approximate Dynamic Programming
- 7 Iterative Algorithms
- 8 Summary & Lookahead
- 9 References

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.