Lecture 03: Dynamic Programming

Paul Swoboda

UNIVERSITÄT MANNHEIM

Table of Contents

What is Dynamic Programming (DP)?

Basic DP definition

- Dynamic: sequential or temporal problem structure
- Programming: mathematical optimization, i.e., numerical solutions

Further characteristics:

- DP is a collection of algorithms to solve MDPs and neighboring problems.
  - We will focus only on finite MDPs.
  - In case of continuous action/state space: apply quantization.
- Use of value functions to organize and structure the search for an optimal policy.
- Breaks problems into subproblems and solves them.

Requirements for DP

DP can be applied to problems with the following characteristics.

- Optimal substructure:
  - Principle of optimality applies.
  - Optimal solution can be derived from subproblems.
- Overlapping subproblems:
  - Subproblems recur many times.
  - Hence, solutions can be cached and reused.

How is that connected to MDPs?

- MDPs satisfy above's properties:
  - Bellman equation provides recursive decomposition.
  - Value function stores and reuses solutions.

Example: DP vs. Exhaustive Search (1)

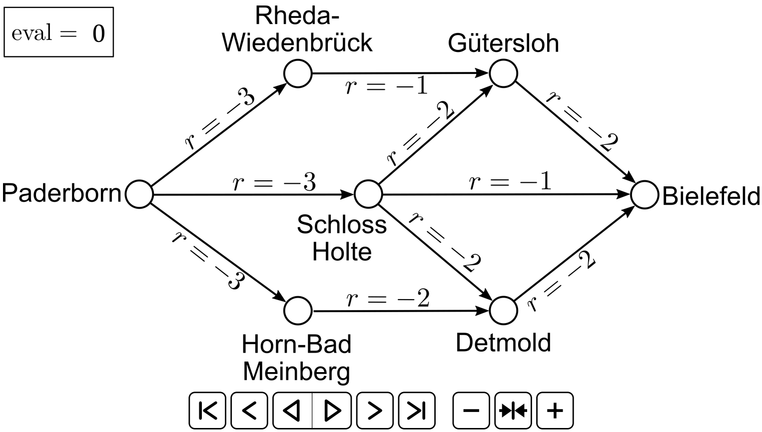


Fig. 1.1: Shortest path problem to travel from Paderborn to Bielefeld: Eshaustive search requires 14 travel segment evaluations since every possible travel route is evaluated independently.

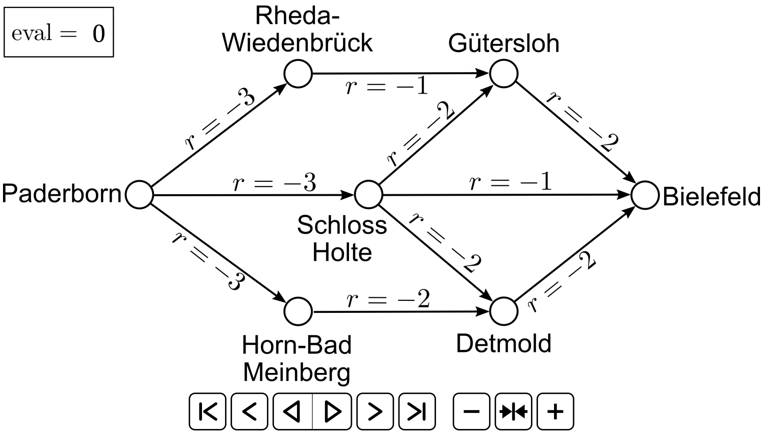


Fig. 1.2: Shortest path problem to travel from Paderborn to Bielefeld: DP requires only 10 travel segment evaluations in order to calculate the optimal travel policy due to the reuse of subproblem results.

Utility of DP in the RL Context

DP is used for iterative planning (i.e., model-based prediction and control) in an MDP.

- Prediction:
  - Input: MDP $\langle \mathcal{S}, \mathcal{A}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$ and policy π
  - Output: (estimated) value function $\hat{v}_\pi \approx v_\pi$
- Control:
  - Input: MDP $\langle \mathcal{S}, \mathcal{A}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$
  - Output: (estimated) optimal value function $\hat{v}_\pi^* \approx v_\pi^*$ or policy $\hat{\pi}^* \approx \pi^*$

In both applications DP requires full knowledge of the MDP structure.

- Feasibility in real-world engineering applications (model vs. system) is therefore limited.
- But: following DP concepts are largely used in modern data-driven RL algorithms.

Table of Contents

Policy Evaluation Background (1)

- Problem: evaluate a given policy π to predict v_π.
- Recap: Bellman equation for $s_k \in \mathcal{S}$ is given as

$$\begin{aligned} v_\pi(s_k) &= \mathbb{E}_\pi\left[G_k | S_k = s_k\right], \\ &= \mathbb{E}_\pi\left[R_{k+1} + \gamma G_{k+1} | S_k = s_k\right], \\ &= \mathbb{E}_\pi\left[R_{k+1} + \gamma v_\pi(S_{k+1}) | S_k = s_k\right]. \end{aligned}$$

- Or in matrix form:

$$\boldsymbol{v}_{\mathcal{S}}^\pi = \boldsymbol{r}_{\mathcal{S}}^\pi + \gamma \boldsymbol{\mathcal{P}}_{ss'}^\pi \boldsymbol{v}_{\mathcal{S}}^\pi,$$

$$\begin{bmatrix} v_1^\pi \\ \vdots \\ v_n^\pi \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1^\pi \\ \vdots \\ \mathcal{R}_n^\pi \end{bmatrix} + \gamma \begin{bmatrix} p_{11}^\pi & \cdots & p_{1n}^\pi \\ \vdots & & \vdots \\ p_{n1}^\pi & \cdots & p_{nn}^\pi \end{bmatrix} \begin{bmatrix} v_1^\pi \\ \vdots \\ v_n^\pi \end{bmatrix}.$$

- Solving the Bellman equation for v_π requires handling a linear equation system with n unknowns (i.e., number of states).
- Remember that the reward function $\mathcal{R}_s^\pi$ might also contain stochastic influences depending on the MDP structure

Policy Evaluation Background (2)

- Problem: directly calculating v_π is numerically costly for high-dimensional state spaces (e.g., by matrix inversion).
- General idea: apply iterative approximations $\hat{v}_i(s_k) = v_i(s_k)$ of $v_\pi(s_k)$ with decreasing errors:

$$\left\| v_i(s_k) - v_\pi \right\|_\infty \to 0 \quad \text{for} \quad i = 1, 2, 3, \ldots \tag{1.1}$$

- The Bellman equation in matrix form can be rewritten as:

$$\underbrace{\left(\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}}_{ss'}^{\pi}\right)}_{\boldsymbol{A}} \underbrace{\boldsymbol{v}_{\mathcal{S}}^{\pi}}_{\boldsymbol{\zeta}} = \underbrace{\boldsymbol{r}_{\mathcal{S}}^{\pi}}_{\boldsymbol{b}} . \tag{1.2}$$

- To iteratively solve this linear equation $\boldsymbol{A}\boldsymbol{\zeta} = \boldsymbol{b}$, one can apply numerous methods such as
  - General gradient descent,
  - Richardson iteration,
  - Krylov subspace methods.

Richardson Iteration (1)

In the MDP context, the Richardson iteration became the default solution approach to iteratively solve:

$$\boldsymbol{A}\boldsymbol{\zeta} = \boldsymbol{b}.$$

The Richardson iteration is

$$\boldsymbol{\zeta}_{i+1} = \boldsymbol{\zeta}_i + \omega(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{\zeta}_i) \tag{1.3}$$

with ω being a scalar parameter that has to be chosen such that the sequence $\boldsymbol{\zeta}_i$ converges. To choose ω we inspect the series of approximation errors $\boldsymbol{e}_i = \boldsymbol{\zeta}_i - \boldsymbol{\zeta}$ and apply it to (1.3):

$$\boldsymbol{e}_{i+1} = \boldsymbol{e}_i - \omega \boldsymbol{A}\boldsymbol{e}_i = (\boldsymbol{I} - \omega \boldsymbol{A})\,\boldsymbol{e}_i. \tag{1.4}$$

To evaluate convergence we inspect the following norm:

$$\|\boldsymbol{e}_{i+1}\|_\infty = \|(\boldsymbol{I} - \omega \boldsymbol{A})\,\boldsymbol{e}_i\|_\infty . \tag{1.5}$$

Richardson Iteration (2)

Since any induced matrix norm is sub-multiplicative, we can approximate (1.5) by the inequality:

$$\left\|\boldsymbol{e}_{i+1}\right\|_\infty \leq \left\|(\boldsymbol{I} - \omega \boldsymbol{A})\right\|_\infty \left\|\boldsymbol{e}_i\right\|_\infty . \tag{1.6}$$

Hence, the series converges if

$$\left\|(\boldsymbol{I} - \omega \boldsymbol{A})\right\|_\infty < 1. \tag{1.7}$$

Inserting from (1.2) leads to:

$$\left\|(\boldsymbol{I}(1-\omega) + \omega\gamma\boldsymbol{\mathcal{P}}_{ss'}^{\pi})\right\|_\infty < 1. \tag{1.8}$$

For $\omega = 1$ we receive:

$$\gamma \left\|(\boldsymbol{\mathcal{P}}_{ss'}^{\pi})\right\|_\infty < 1. \tag{1.9}$$

Since the row elements of $\boldsymbol{\mathcal{P}}_{ss'}^{\pi}$ always sum up to 1,

$$\gamma < 1 \tag{1.10}$$

follows. Hence, when discounting the Richardson iteration always converges for MDPs even if we assume $\omega = 1$.

Iterative Policy Evaluation by Richardson Iteration (1)

General form for any $s_k \in \mathcal{S}$ at iteration i is given as:

$$v_{i+1}(s_k) = \sum_{a_k \in \mathcal{A}} \boldsymbol{\pi}(a_k|s_k) \left(\mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a v_i(s_{k+1}) \right) . \qquad (1.11)$$

Matrix form then is:

$$\boldsymbol{v}_{\mathcal{S},i+1}^{\pi} = \boldsymbol{r}_{\mathcal{S}}^{\pi} + \gamma \boldsymbol{\mathcal{P}}_{ss'}^{\pi} \boldsymbol{v}_{\mathcal{S},i}^{\pi} . \qquad (1.12)$$

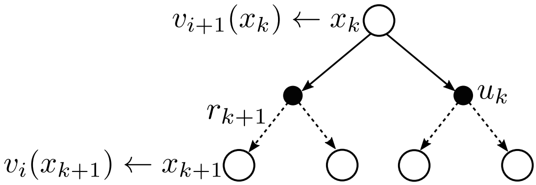


Fig. 1.3: Backup diagram for iterative policy evaluation

Iterative Policy Evaluation by Richardson Iteration (2)

- During one Richardson iteration the 'old' value of s_k is replaced with a 'new' value from the 'old' values of the successor state s_{k+1}.
  - Update $v_{i+1}(s_k)$ from $v_i(s_{k+1})$, see Fig. 1.3.
  - Updating estimates (v_{i+1}) on the basis of other estimates (v_i) is often called bootstrapping.
- The Richardson iteration can be interpreted as a gradient descent algorithm for solving (1.2).
- This leads to synchronous, full backups of the entire state space $\mathcal{S}$.
- Also called expected update because it is based on the expectation over all possible next states (utilizing full knowledge).
- In subsequent lectures, the expected update will be supplemented by data-driven samples from the environment.

Iterative Policy Evaluation Example: Forest Tree MDP

Let's reuse the forest tree MDP example with *fifty-fifty policy* and discount factor $\gamma = 0.8$ plus disaster probability $\alpha = 0.2$:

$$\boldsymbol{\mathcal{P}}_{ss'}^{\pi} = \begin{bmatrix} 0 & \frac{1-\alpha}{2} & 0 & \frac{1+\alpha}{2} \\ 0 & 0 & \frac{1-\alpha}{2} & \frac{1+\alpha}{2} \\ 0 & 0 & \frac{1-\alpha}{2} & \frac{1+\alpha}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{r}_{\mathcal{S}}^{\pi} = \begin{bmatrix} 0.5 \\ 1 \\ 2 \\ 0 \end{bmatrix}.$$

i	$v_i(s=1)$	$v_i(s=2)$	$v_i(s=3)$	$v_i(s=4)$
0	0	0	0	0
1	0.5	1	2	0
2	0.82	1.64	2.64	0
3	1.03	1.85	2.85	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
∞	1.12	1.94	2.94	0

Tab. 1.1: Policy evaluation by Richardson iteration (1.12) for forest tree MDP

Variant: In-Place Updates

Instead of applying (1.12) to the entire vector $\boldsymbol{v}^{\pi}_{\mathcal{S},i+1}$ in 'one shot' (synchronous backup), an elementwise in-place version of the policy evaluation can be carried out:

input: full model of the MDP, i.e., $\langle \mathcal{S}, \mathcal{A}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$ including policy π
parameter: $\delta > 0$ as accuracy termination threshold
init: $v_0(s)\,\forall\, s \in \mathcal{S}$ arbitrary except $v_0(s) = 0$ if s is terminal
repeat
    $\Delta \leftarrow 0$;
    for $\forall\, s_k \in \mathcal{S}$ **do**
        $\tilde{v} \leftarrow \hat{v}(s_k)$;
        $\hat{v}(s_k) \leftarrow \sum_{a_k \in \mathcal{A}} \pi(a_k|s_k) \left(\mathcal{R}^a_s + \gamma \sum_{s_{k+1} \in \mathcal{S}} p^a_{ss'} \hat{v}(s_{k+1}) \right)$;
        $\Delta \leftarrow \max\left(\Delta, |\tilde{v} - \hat{v}(s_k)|\right)$;
until $\Delta < \delta$;

Algo. 1.1: Iterative policy evaluation using in-place updates (output: estimate of $\boldsymbol{v}^{\pi}_{\mathcal{S}}$)

In-Place Policy Evaluation Updates for Forest Tree MDP

- In-place algorithms allow to update states in a beneficial order.
- May converge faster than regular Richardson iteration if state update order is chosen wisely (sweep through state space).
- For forest tree MDP: reverse order, i.e., start with $x = 4$.
- As can be seen in Tab. 1.2 the in-place updates especially converge faster for the 'early states'.

i	$v_i(x=1)$	$v_i(x=2)$	$v_i(x=3)$	$v_i(x=4)$
0	0	0	0	0
1	1.03	1.64	2	0
2	1.09	1.85	2.64	0
3	1.11	1.91	2.85	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
∞	1.12	1.94	2.94	0

Tab. 1.2: In-place updates for forest tree MDP

Table of Contents

General Idea on Policy Improvement

- If we know v_π of a given MDP, how to improve the policy?
- The simple idea of policy improvement is:
  - Consider a new (non-policy conform) action $a \neq \pi(s_k)$.
  - Follow thereafter the current policy π.
  - Check the action-value of this 'new move'. If it is better than the 'old' value, take it.

$$q_\pi(s_k, a_k) = \mathbb{E}\left[R_{k+1} + \gamma v_\pi(S_{k+1}) | S_k = s_k, A_k = a_k\right] . \tag{1.13}$$

Theorem 1.1: Policy improvement

If for any deterministic policy pair π and π'

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \quad \forall s \in \mathcal{S} \tag{1.14}$$

applies, then the policy π' must be as good as or better than π. Hence, it obtains greater or equal expected return

$$v_{\pi'}(s) \geq v_\pi(s) \quad \forall s \in \mathcal{S}. \tag{1.15}$$

Greedy Policy Improvement (1)

- So far, policy improvement addressed only changing the policy at a single state.
- Now, extend this scheme to all states by selecting the best action according to $q_\pi(s_k, a_k)$ in every state (greedy policy improvement):

$$\begin{aligned}
\pi'(s_k) &= \arg\max_{a_k \in \mathcal{A}} q_\pi(s_k, a_k), \\
&= \arg\max_{a_k \in \mathcal{A}} \mathbb{E}\left[R_{k+1} + \gamma v_\pi(S_{k+1}) | S_k = s_k, A_k = a_k\right], \\
&= \arg\max_{a_k \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a v_\pi(s_{k+1})\,.
\end{aligned} \tag{1.16}$$

Greedy Policy Improvement (2)

- Each greedy policy improvement takes the best action in a one-step look-ahead search and, therefore, satisfies Theo. 1.1.
- If after a policy improvement step $v_\pi(s_k) = v_{\pi'}(s_k)$ applies, it follows:

$$\begin{aligned} v_{\pi'}(s_k) &= \max_{a_k \in \mathcal{A}} \mathbb{E}\left[R_{k+1} + \gamma v_{\pi'}(S_{k+1}) | S_k = s_k, A_k = a_k\right], \\ &= \max_{a_k \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^a v_{\pi'}(s_{k+1}). \end{aligned} \tag{1.17}$$

- This is the Bellman optimality equation, which guarantees that $\pi' = \pi$ must be optimal policies.
- Although proof for policy improvement theorem was presented for deterministic policies, transfer to stochastic policies $\pi(a_k|s_k)$ is possible.
- Takeaway message: policy improvement theorem guarantees finding optimal policies in finite MDPs (e.g., by DP).

Table of Contents

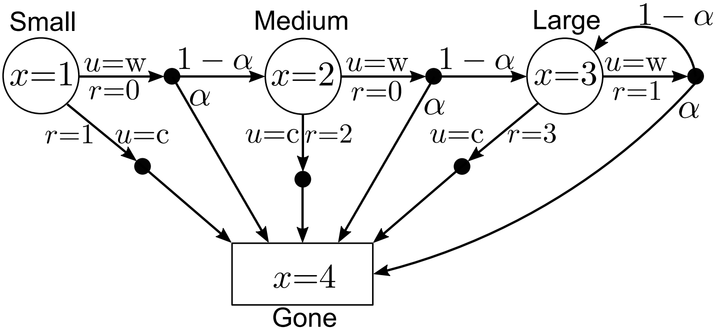
Concept of Policy Iteration

- Policy iteration combines the previous policy evaluation and policy improvement in an iterative sequence:

$$\pi_0 \rightarrow v_{\pi_0} \rightarrow \pi_1 \rightarrow v_{\pi_1} \rightarrow \cdots \pi^* \rightarrow v_{\pi^*} \tag{1.18}$$

- Evaluate $\rightarrow$ improve $\rightarrow$ evaluate $\rightarrow$ improve ...
- In the 'classic' policy iteration, each policy evaluation step in (1.18) is fully executed, i.e., for each policy π_i an exact estimate of v_{π_i} is provided either by iterative policy evaluation with a sufficiently high number of steps or by any other method that fully solves (1.2).

Policy Iteration Example: Forest Tree MDP (1)



- Two actions possible in each state:
  - Wait $a = w$: let the tree grow.
  - Cut $a = c$: gather the wood.

Policy Iteration Example: Forest Tree MDP (2)

Assume $\alpha = 0.2$ and $\gamma = 0.8$ and start with 'tree hater' initial policy:

1. $\pi_0 = \pi(a_k = \mathsf{c}|s_k) \quad \forall s_k \in \mathcal{S}$.
2. Policy evaluation: $v_{\mathcal{S}}^{\pi_0} = \begin{bmatrix}1 & 2 & 3 & 0\end{bmatrix}^T$
3. Greedy policy improvement:

$$\begin{aligned}\pi_1(s_k) &= \arg\max_{a_k \in \mathcal{A}} \mathbb{E}\left[R_{k+1} + \gamma v_{\pi_0}(S_{k+1})|S_k = s_k, A_k = a_k\right], \\ &= \{\pi(a_k = \mathsf{w}|s_k = 1), \pi(a_k = \mathsf{c}|s_k = 2), \pi(a_k = \mathsf{c}|s_k = 3)\}\end{aligned}$$

4. Policy evaluation: $v_{\mathcal{S}}^{\pi_1} = \begin{bmatrix}1.28 & 2 & 3 & 0\end{bmatrix}^T$
5. Greedy policy improvement:

$$\begin{aligned}\pi_2(s_k) &= \arg\max_{a_k \in \mathcal{A}} \mathbb{E}\left[R_{k+1} + \gamma v_{\pi_1}(S_{k+1})|S_k = s_k, A_k = a_k\right], \\ &= \{\pi(a_k = \mathsf{w}|s_k = 1), \pi(a_k = \mathsf{c}|s_k = 2), \pi(a_k = \mathsf{c}|s_k = 3)\}, \\ &= \pi_1(s_k) \\ &= \pi^*\end{aligned}$$

Policy Iteration Example: Forest Tree MDP (3)

Assume $\alpha = 0.2$ and $\gamma = 0.8$ and start with 'tree lover' initial policy:

1. $\pi_0 = \pi(a_k = \mathsf{w}|s_k) \quad \forall s_k \in \mathcal{S}$.
2. Policy evaluation: $v_{\mathcal{S}}^{\pi_0} = \begin{bmatrix} 1.14 & 1.78 & 2.78 & 0 \end{bmatrix}^T$
3. Greedy policy improvement:

$$\begin{aligned} \pi_1(s_k) &= \arg\max_{a_k \in \mathcal{A}} \mathbb{E}\left[R_{k+1} + \gamma v_{\pi_0}(S_{k+1})|S_k = s_k, A_k = a_k\right], \\ &= \{\pi(a_k = \mathsf{w}|s_k = 1), \pi(a_k = \mathsf{c}|s_k = 2), \pi(a_k = \mathsf{c}|s_k = 3)\} \end{aligned}$$

4. Policy evaluation: $v_{\mathcal{S}}^{\pi_1} = \begin{bmatrix} 1.28 & 2 & 3 & 0 \end{bmatrix}^T$
5. Greedy policy improvement:

$$\begin{aligned} \pi_2(s_k) &= \arg\max_{a_k \in \mathcal{A}} \mathbb{E}\left[R_{k+1} + \gamma v_{\pi_1}(S_{k+1})|S_k = s_k, A_k = a_k\right], \\ &= \{\pi(a_k = \mathsf{w}|s_k = 1), \pi(a_k = \mathsf{c}|s_k = 2), \pi(a_k = \mathsf{c}|s_k = 3)\}, \\ &= \pi_1(s_k) \\ &= \pi^* \end{aligned}$$

Policy Iteration Example: Jack's Car Rental (1)

- States: Two rental locations, maximum of 20 cars each
- Actions: Move up to 5 cars between locations overnight
- Reward:
  - $+10$ \$ for each car rented (if available at location)
  - -2 \$ for each overnight car transfer
  - Discount: $\gamma = 0.9$
- Dynamics: Cars returned and requested randomly following Poisson distribution
  - $P_\lambda(n) = \frac{\lambda^n}{n!} e^{-\lambda}$
  - $P_\lambda(n) =$ probability of observing n events with mean event rate λ
  - 1st location: $\lambda_{\text{req.}} = 3$, $\lambda_{\text{ret.}} = 3$
  - 2nd location: $\lambda_{\text{req.}} = 4$, $\lambda_{\text{ret.}} = 2$

Policy Iteration Example: Jack's Car Rental (2)

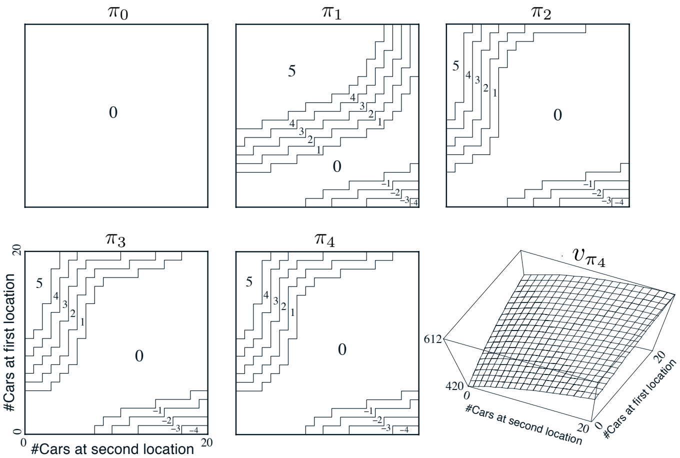


Fig. 1.4: Sequence of policies found by policy iteration including optimal state value after termination (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Value Iteration (1)

- Policy iteration involves full policy evaluation steps between policy improvements.
- In large state-space MDPs the full policy evaluation may be numerically very costly.
- Using a limited number of iterative policy evaluations steps and then apply policy improvement may speed up the entire DP process.
- Value iteration: One step iterative policy evaluation followed by policy improvement.
- Allows simple update rule which combines policy improvement with truncated policy evaluation:

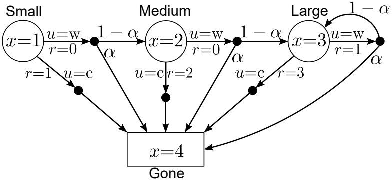
$$
\begin{aligned}
v_{i+1}(s_k) &= \max_{a_k \in \mathcal{A}} \mathbb{E}\left[R_{k+1} + \gamma v_i(S_{k+1}) | S_k = s_k, A_k = a_k\right], \\
&= \max_{a_k \in \mathcal{A}} \mathcal{R}_x^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^u v_i(s_{k+1}).
\end{aligned}
\tag{1.19}
$$

Value Iteration (2)

input: full model of the MDP, i.e., $\langle \mathcal{S}, \mathcal{A}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$
parameter: $\delta > 0$ as accuracy termination threshold
init: $v_0(x) \, \forall \, x \in \mathcal{S}$ arbitrary except $v_0(x) = 0$ if x is terminal
repeat
    $\Delta \leftarrow 0$;
    for $\forall \, s_k \in \mathcal{S}$ **do**
        $\tilde{v} \leftarrow \hat{v}(s_k)$;
        $\hat{v}(s_k) \leftarrow \max_{a_k \in \mathcal{A}} \left(\mathcal{R}_x^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^u \hat{v}(s_{k+1}) \right)$;
        $\Delta \leftarrow \max \left(\Delta, |\tilde{v} - \hat{v}(s_k)| \right)$;
until $\Delta < \delta$;
output: Deterministic policy $\pi \approx \pi^*$, such that
$\pi(s_k) \leftarrow \arg\max_{a_k \in \mathcal{A}} \left(\mathcal{R}_x^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^u \hat{v}(s_{k+1}) \right)$;

Algo. 1.2: Value iteration (note: compared to policy iteration, value iteration doesn't require an initial policy but only a state-value guess)

Value Iteration for Forest Tree MDP



- Assume again $\alpha = 0.2$ and $\gamma = 0.8$.
- Similar to in-place update policy evaluation, reverse order and start value iteration with $x = 4$.
- As shown in Tab. 1.3 value iteration converges in one step (for the given problem) to the optimal state-value.

i	$v_i(x = 1)$	$v_i(x = 2)$	$v_i(x = 3)$	$v_i(x = 4)$
0	0	0	0	0
1	1.28	2	3	0
*	1.28	2	3	0

Tab. 1.3: Value iteration for forest tree MDP

Table of Contents

Summarizing DP Algorithms

- All DP algorithms are based on the state-value $v(x)$.
  - Complexity is $\mathcal{O}(m \cdot n^2)$ for m actions and n states.
  - Evaluate all n^2 state transitions while considering up to m actions per state.
- Could be also applied to action-values $q(x, u)$.
  - Complexity is inferior with $\mathcal{O}(m^2 \cdot n^2)$.
  - There are up to m^2 action-values which require n^2 state transition evaluations each.

Problem	Relevant Equations	Algorithm
prediction	Bellman expectation eq.	policy evaluation
control	Bellman expectation eq. & greedy policy improvement	policy iteration
control	Bellman optimality eq.	value iteration

Tab. 1.4: Short overview addressing the treated DP algorithms

Asynchronous DP

- DP algorithms considered so far used synchronous backups:
  - In one iteration the entire state space is updated.
  - May be computational expensive for large MDPs.
  - Some state-values or policy parts may converge faster than other but are updated as often as slowly converging states.
- In contrast, asynchronous backups update states individually in an (arbitrary) order:
  - Choose smart order to achieve faster overall convergence rate.
  - Some states may be updated more frequently than others.
  - Overall algorithms converges if all states are still visited to some extent (important requirement to ensure convergence).
  - Simple example: in-place policy evaluation where only a subset of all states are updated each iterations (cf. Algo. 1.1).

Asynchronous DP: Prioritized Sweeping

- Use magnitude of Bellman error as an indicator which state should be updated next:

$$\arg\max_{s_k \in \mathcal{S}} \left| \max_{a_k \in \mathcal{A}} \left(\mathcal{R}_x^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^u v_i(s_{k+1}) \right) - v_i(s_k) \right| . \tag{1.20}$$

- Update the state with the largest Bellman error first.
- Build up a priority queue of most relevant states by refreshing the Bellman error after each state update.

Asynchronous DP: Real-Time Updates

- Update those states which are frequently visited by the agent.
- Utilizes agent's experience to guide the asynchronous DP updates.
- After each time step $\langle s_k, a_k, r_{k+1}\rangle$ update s_k:

$$v_i(s_k) \leftarrow \max_{a_k \in \mathcal{A}} \left(\mathcal{R}_x^a + \gamma \sum_{s_{k+1} \in \mathcal{S}} p_{ss'}^u v_i(s_{k+1}) \right). \tag{1.21}$$

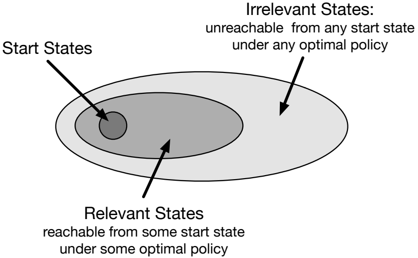
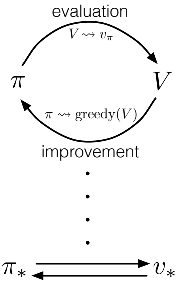


Fig. 1.5: Real-time DP updates focus on reachable states (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Generalized Policy Iteration (GPI)

- Almost all RL methods are well-described as GPI.
- Push-pull: Improving the policy will deteriorate value estimation.
- Well balanced trade-off between evaluating and improving is required.



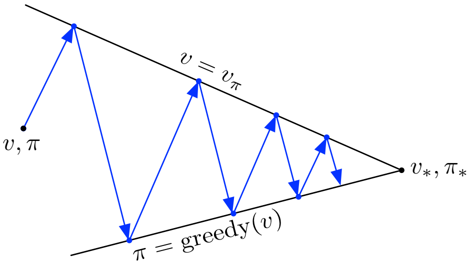


Fig. 1.6: Interpreting generalized policy iteration to switch back and forth between (arbitrary) evaluations and improvement steps (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Curse of Dimensionality

- DP is much more efficient than an exhaustive search over all n states and m actions in finite MDPs in order to find an optimal policy.
  - Exhaustive search for deterministic policies: m^n evaluations.
  - DP results in polynomial complexity regarding m and n.
- Nevertheless, DP uses full-width backups:
  - For each state update, every successor state and action is considered.
  - While utilizing full knowledge of the MDP structure.
- Hence, DP can be effective up to medium-sized MDPs (i.e., million states)
- For large problems DP suffers from the curse of dimensionality:
  - Number of finite states n grows exponentially with the number of state variables.
  - Also: if continuous variables need quantization typically a large number of states results.
  - Single state update may become computational infeasible.

Summary: What You've Learned Today

- DP is applicable for prediction and control problems in MDPs.
- But requires always full knowledge about the environment (i.e., it is a model-based solution also called planning).
- DP is more efficient than exhaustive search.
- But suffers from the curse of dimensionality for large MDPs.
- (Iterative) policy evaluations and (greedy) improvements solve MDPs.
- Both steps can be combined via value iteration.
- This idea of (generalized) policy iteration is a basic scheme of RL.
- Implementing DP algorithms comes with many degrees of freedom.
- For example how to order the state updates (asyn. vs. sync.).