HSLU Lucerne University of Applied Sciences and Arts

Dynamic Programming

Reinforcement Learning
October 13, 2022

FH Zentralschweiz

Learning Objectives

Understand the Bellman equation (better ☺)

Distinguish between policy evaluation and control

Explain where dynamic programming can be used and where not

Explain how to compute value functions using policy iteration

Understand policy improvement

...

Classical Dynamic Programming

Some basic ideas for the use of (classical) dynamic programming are:

- Divide and conquer
- Subdivide larger problems into smaller problems
- Solve smaller problems just once (and store solution)
- Make decisions in stages

Example: 0-1 Knapsack Problem

Given a set of items with weights and values, pack a knapsack with given maximal weight to contain items of maximal total value

```
# values and corresponding weights
v = [20, 5, 10, 40, 15, 25]
w = [1, 2, 3, 8, 7, 4]

# maximal weight
W = 10
```

Knapsack Problem

Solution:

- Recursively calculate a solution with/without the item
- Take the maximal value of both

```python
def knapsack(v, w, n, W):
    """
    v: values, w: weights, n: item to consider, W: weight left
    """
    if W < 0:
        return -sys.maxsize
    if n < 0 or W == 0:
        return 0
    include = v[n] + knapsack(v, w, n - 1, W - w[n])
    exclude = knapsack(v, w, n - 1, W)
    return max(include, exclude)
```

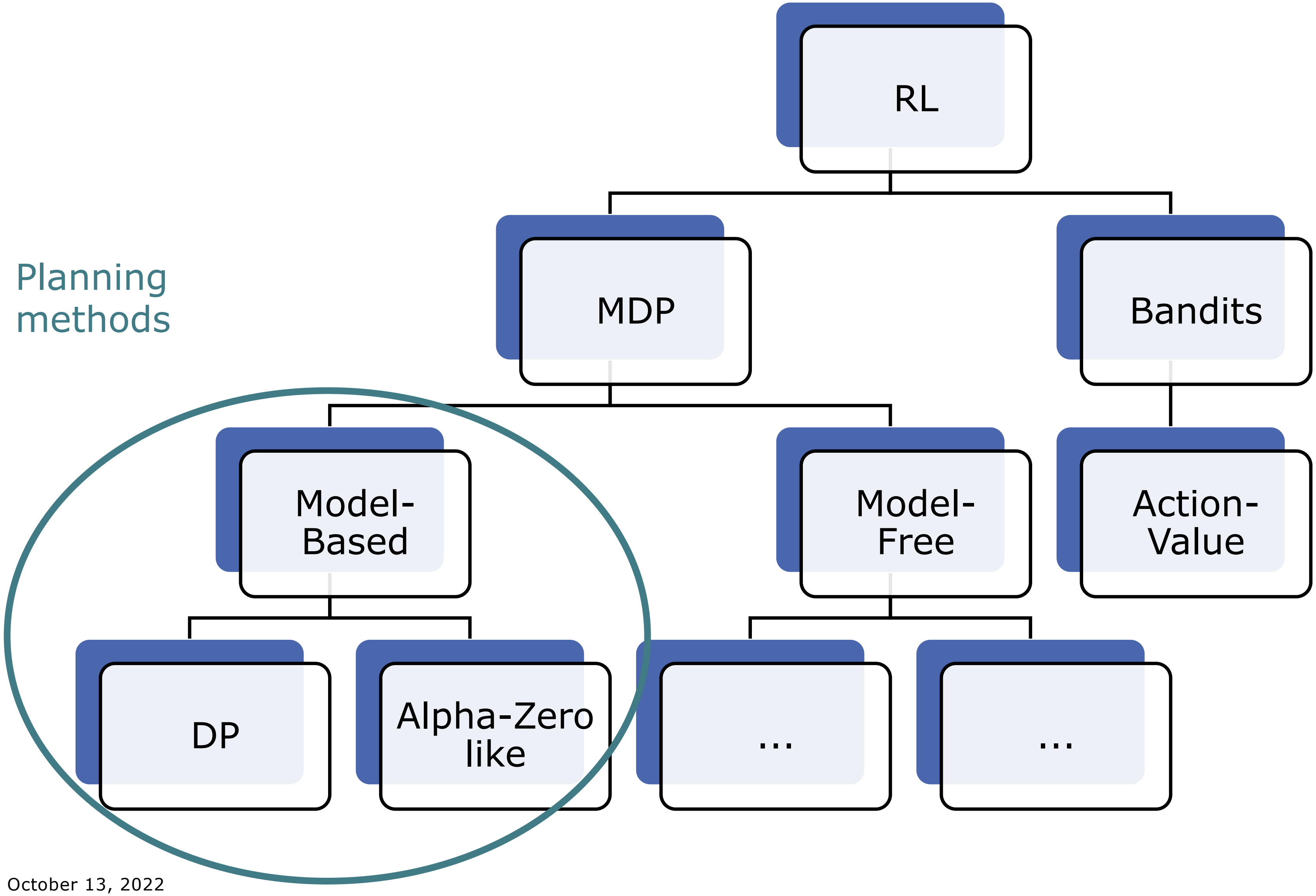
Dynamic Programming in RL

Key ideas:

- Algorithms to compute optimal value functions and policies given a perfect **model** of the MDP
- Finite MDP environment
- Use value functions to organize the search for good policies
- An optimal policy can be obtained from an optimal value function
- DP is often limited in RL due to the need of a model and the large computational expense

RL Methods

Planning methods



Bellman Equation

$$v_\pi(s) \;=\; \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma v_\pi(s')\big], \;\text{ for all } s \in \mathcal{S}$$

- Calculates the value of a state by following policy π
- Could be solved for v_π using a system of linear equations, but…
- ... iterative solutions are usually preferred (as they are computationally more efficient)

Policy Evaluation (Prediction)

- Given a policy, what is its value function?

- Iterative computation using the Bellman equation
- Calculate a sequence of values that approach the correct solution

$$
\begin{aligned}
v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_k(s')\right]
\end{aligned}
$$

- This is called *iterative policy evaluation*.

Iterative Policy Evaluation, estimate v_π

Input: a policy π

Initialize:

  $V(s) \in \mathbb{R}$ arbitrarily, except $V(\text{terminal}) = 0$

Loop:

  $\Delta \leftarrow 0$

  Loop for each $s \in \mathcal{S}$:

    $v \leftarrow V(s)$

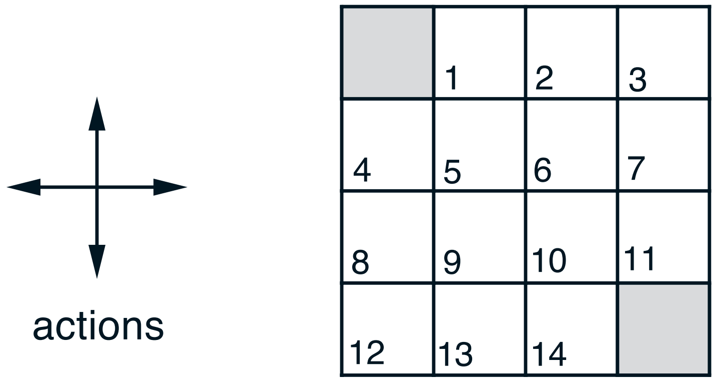
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V(s')\right]$

    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

  until $\Delta < \theta$

Example for Policy Evaluation

- Small *Gridworld* Example
- Find terminal state (grey) from any position in minimal number of steps
- Start with random policy
- Initialize all values with 0 (any other value is possible too, except for terminal states which must be 0)



$R_t = -1$
on all transitions

Policy Evaluation

- The iterative policy evaluation calculates increasingly better estimates of the value function
- (Example from the book, values are given in only 1 digit precision)

v_k for the Random Policy

$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

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

Policy Improvement

What happens if we change an action a, but follow π otherwise?

Recall that

$$q_\pi(s,a) = \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right]$$

If π and π' are deterministic policies with

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

(i.e. we take the first action from π' then follow π), then

$$v_{\pi'}(s) \geq v_\pi(s), \qquad \forall s \in \ S$$

Policy Improvement

So, if we take a greedy action a, this defines a new policy

$$
\begin{aligned}
\pi'(s) &\doteq \underset{a}{\operatorname{argmax}}\, q_\pi(s,a) \\
&= \underset{a}{\operatorname{argmax}}\, \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\
&= \underset{a}{\operatorname{argmax}} \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s')\right]
\end{aligned}
$$

which is as good as, or better than the old policy π

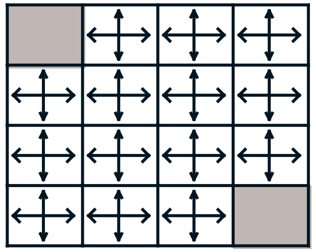
Policy Improvement

v_k for the Random Policy

Greedy Policy w.r.t. v_k

$k = 0$

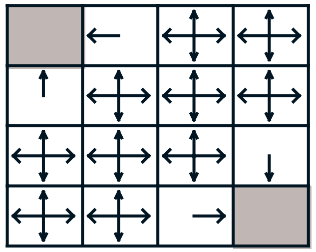
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0



random policy

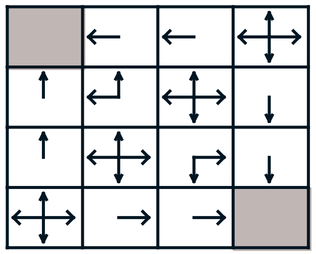
$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0



$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0



Policy Improvement

$k = 3$

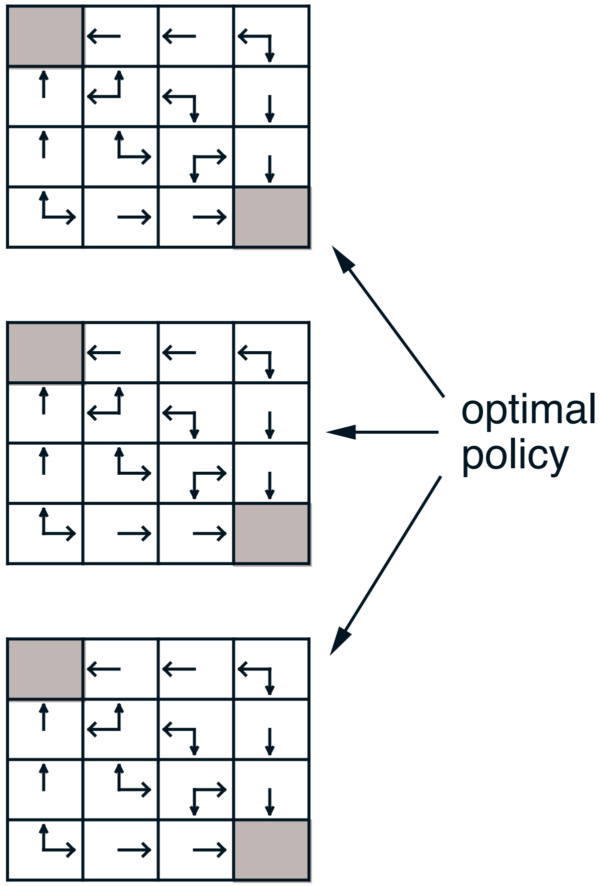
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

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

$k = \infty$

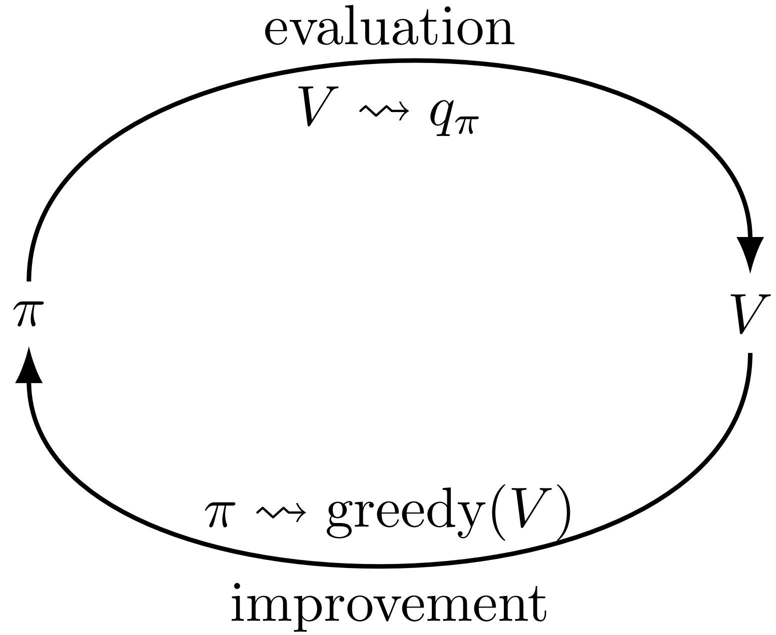
0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0



Policy Iteration

Once a policy π has been improved using v_π to yield a better policy π', we can compute $v_{\pi'}$ and improve it again:

$$\pi_0 \xrightarrow{\mathrm{E}} v_{\pi_0} \xrightarrow{\mathrm{I}} \pi_1 \xrightarrow{\mathrm{E}} v_{\pi_1} \xrightarrow{\mathrm{I}} \pi_2 \xrightarrow{\mathrm{E}} \cdots \xrightarrow{\mathrm{I}} \pi_* \xrightarrow{\mathrm{E}} v_*$$



Policy Iteration

Policy Iteration, estimate $\pi \approx \pi_*$

Initialize:
  $V(s) \in \mathbb{R}$ and $\pi(s)$ arbitrarily, except $V(\text{terminal}) = 0$
Loop:
  Policy Evaluation:
    estimate $V \approx v_\pi$ (see previous algorithm)
  Policy Improvement:
    $\textit{policy-stable} \leftarrow \textit{true}$
    For each $s \in \mathcal{S}$:
      $\textit{old-action} \leftarrow \pi(s)$
      $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s'} p(s', r|s, a)[r + \gamma V(s')]$
      if $\textit{old-action} \neq \pi(s)$, then $\textit{policy-stable} \leftarrow \textit{false}$
until $\textit{policy-stable}$
return $V \approx v_*, \pi \approx \pi_*$

Value Iteration

- Does the policy evaluation need to converge? (see gridworld example) ?
- If we stop policy evaluation after one sweep, we obtain an algorithm called *value iteration*
- The update for this can be written as

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_k(s')\right], \qquad \forall s \in \mathcal{S}$$

Value Iteration

Value Iteration, estimate $\pi \approx \pi_*$

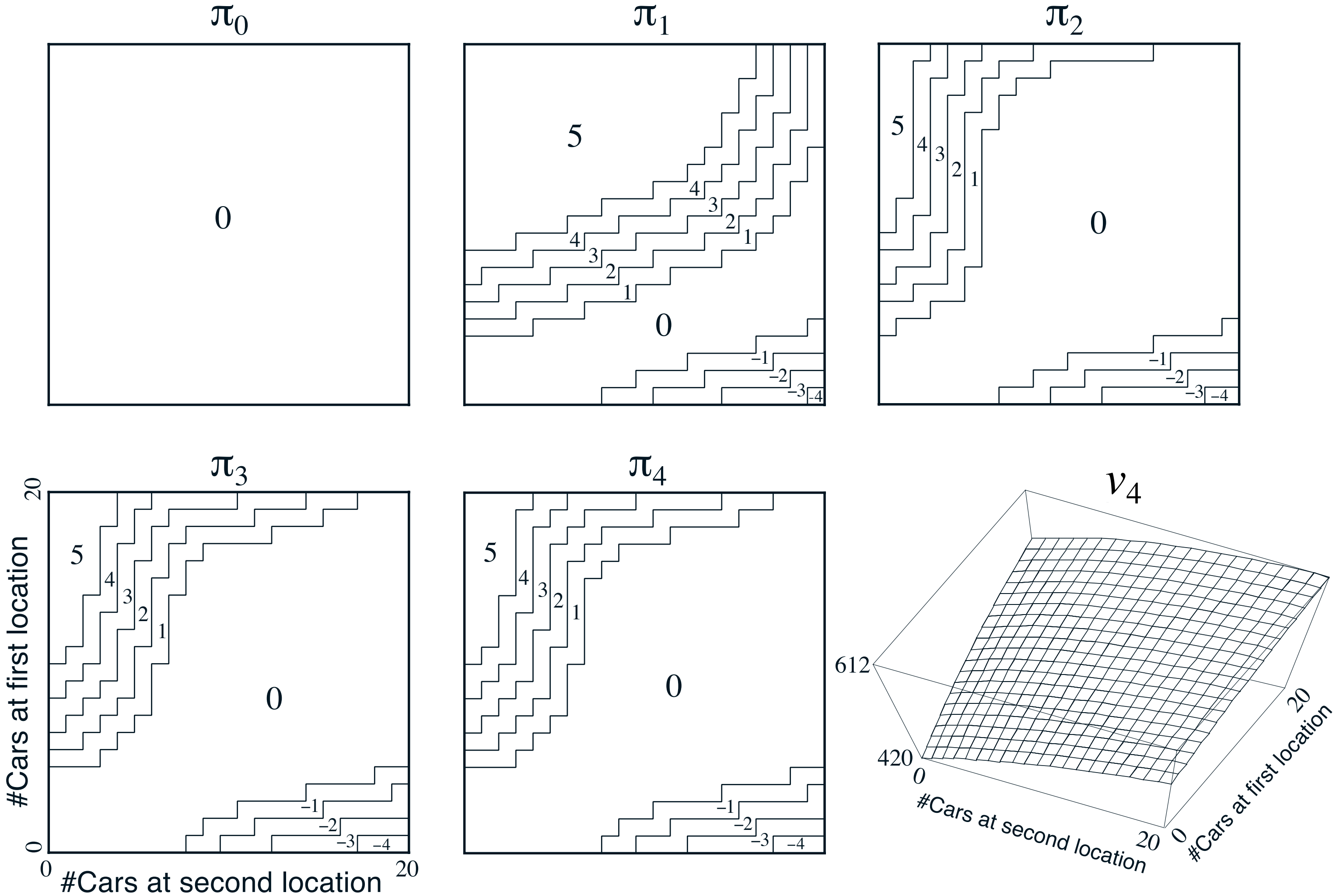
Initialize:
    $V(s) \in \mathbb{R}$ arbitrarily, except $V(\text{terminal}) = 0$
Loop:
    $\Delta \leftarrow 0$
    Loop for each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\,[r + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    until $\Delta < \theta$
Output deterministic policy, such that
    $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\,[r + \gamma V(s')]$

Example: Jack's Car Rental

- 2 car rental locations with max. 20 cars each
- Cars do not have to be returned to the same place
- If cars are available, they can be rented out for $10, if a customer arrives
- Jack can move up to 5 cars during the night at a cost of $2 (this is the action)

- (Poisson model of how the cars are rented and returned, not equal in both locations)
- Policy: Move n cars from first to second location
- What is the optimal policy? I.e., how many cars should be moved in each different state?

Example: Jack's Car Rental



Asynchronous vs. Synchronous Programming

Synchronous DP:

- Update value function in sweeps
- All new values are calculated from the old values
- (Generally requires 2 arrays: one for the old values and one for the new ones)

Asynchronous DP:

- Update value function in any order
- Update values in place
- All new values are calculated from the current values

Generalized Policy Iteration

Generalized Policy Iteration (GPI):

- Interaction between *policy-evaluation* and *policy-improvement*
- *(independent of the granularity of the processes, i.e., if they are run just for one step or until convergence)*
- Many reinforcement learning methods can be described as GPI