(Planning with) Dynamic Programming

DAVIDE BACCIU – BACCIU@DI.UNIPI.IT

IN SUPREMÆ DIGNITATIS 1343
UNIVERSITÀ DI PISA

Introduction

Outline

- ✓ Introduction
- ✓ Dynamic programming
- ✓ Policy Evaluation
- ✓ Policy Iteration
- ✓ Value Evaluation
- ✓ Advanced topics
  - ✓ Asynchronous update
  - ✓ Approximated approaches

What is dynamic programming

Dynamic $\mapsto$ problem with sequential or temporal component

Programming $\mapsto$ optimising a program, i.e. a policy

- ✓A method for solving complex problems by breaking them down into subproblems
  - ✓Solve the subproblems
  - ✓Combine solutions to subproblems
- ✓It is not divide-et-impera
  - ✓Differentiates by overlapping breakdown

Requirements for dynamic programming

- ✓ Optimal substructure
  - ✓ Principle of optimality applies
  - ✓ Optimal solution can be decomposed into subproblems
- ✓ Overlapping subproblems
  - ✓ Subproblems recur many times
  - ✓ Solutions can be cached and reused

Markov decision processes satisfy both properties

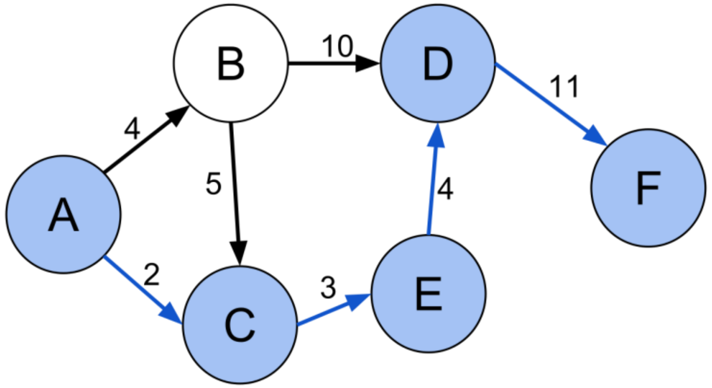
- ✓ Bellman equation gives recursive decomposition
- ✓ Value function stores and reuses solutions

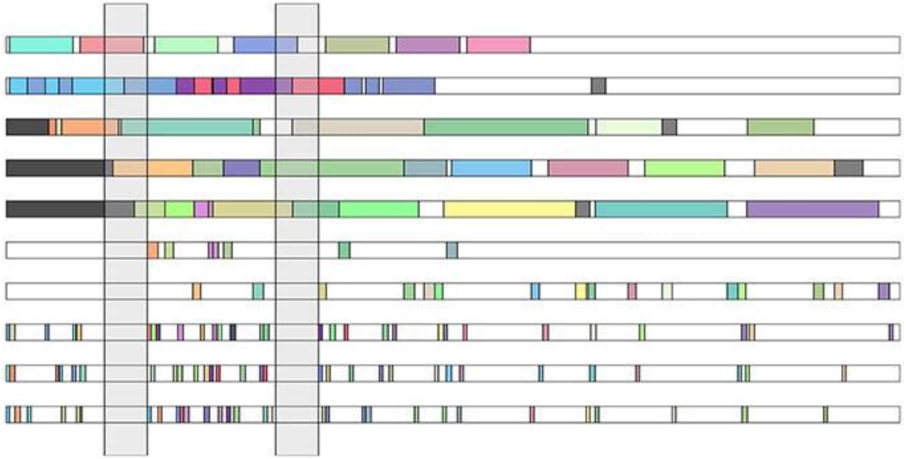
Planning by dynamic programming

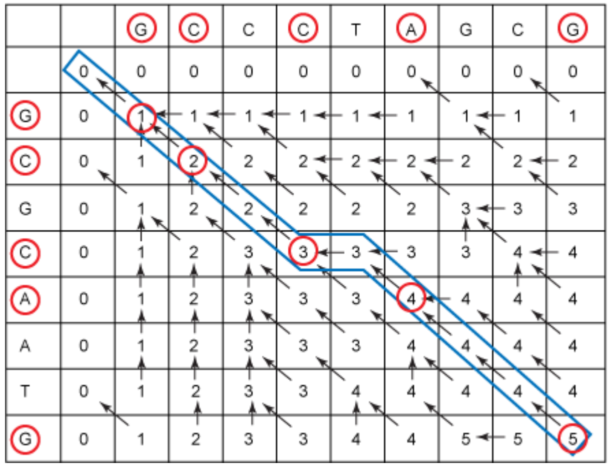
- Dynamic programming assumes full knowledge of the MDP
- Planning in RL (repetita)
  - A model of the environment is known
  - The agent improves its policy

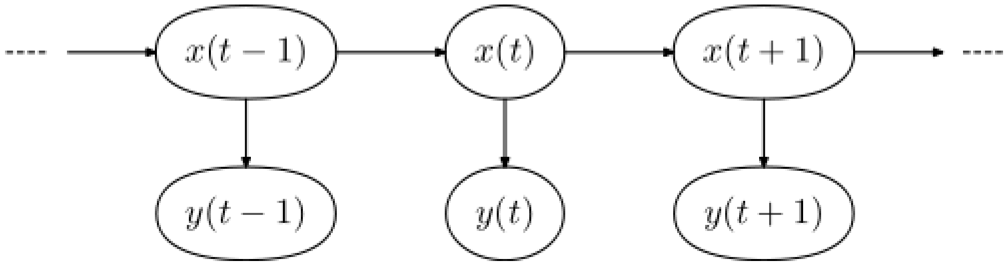
- Dynamic programming can be used for planning in RL
- Prediction
  - Input: MDP $\langle \mathcal{S}, \mathcal{A}, \boldsymbol{P}, \mathcal{R}, \gamma \rangle$ and policy π **or** MRP $\langle \mathcal{S}, \boldsymbol{P}, \mathcal{R}, \gamma \rangle$
  - Output: value function v_π
- Control
  - Input: MDP $\langle \mathcal{S}, \mathcal{A}, \boldsymbol{P}, \mathcal{R}, \gamma \rangle$
  - Output: optimal value function v_{π_*} **and** optimal policy π_*

Applications of Dynamic Programming









Policy Evaluation

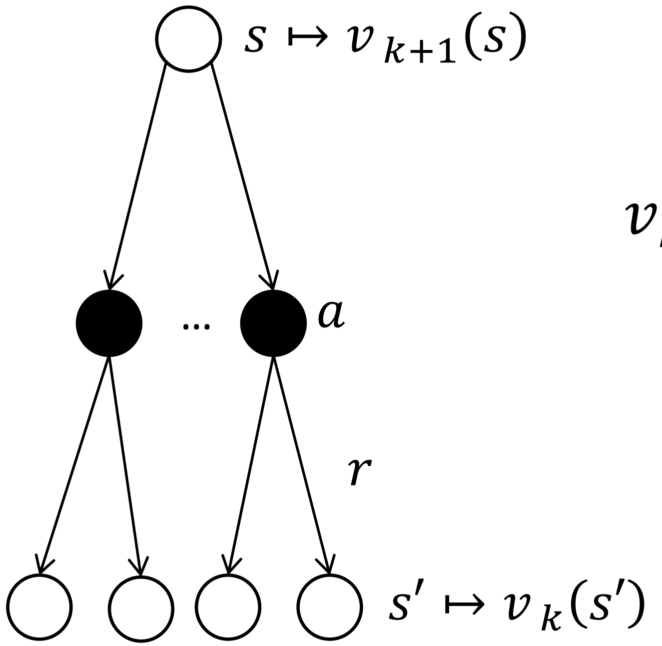
Iterative Policy Evaluation

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

$$v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_\pi$$

- ✓ Using synchronous backups
  i. At each iteration $k + 1$
  ii. For all states $s \in \mathcal{S}$
  iii. Update $v_{k+1}(s)$ from $v_k(s')$ where s' is a successor state of s

Iterative Policy Evaluation - Formally



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

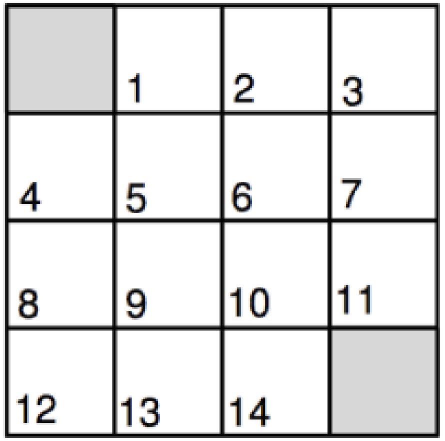
$$v_{k+1} = \mathcal{R}^\pi + \gamma \boldsymbol{P}^{\boldsymbol{\pi}} v_k$$

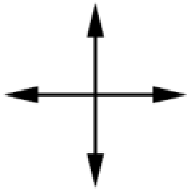
Evaluating a Random Policy in the Small Gridworld

- ✓Undiscounted episodic MPD ($\gamma = 1$)
- ✓Nonterminal states $1, \ldots, 14$
- ✓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

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

r=1 on all transitions





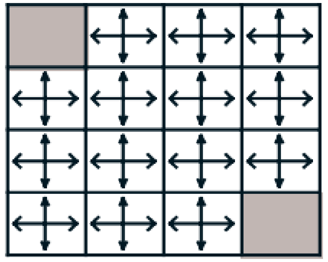
actions

Iterative Policy Evaluation on Small Gridworld (I)

v_k | Greedy policy on v_k

$k = 0$

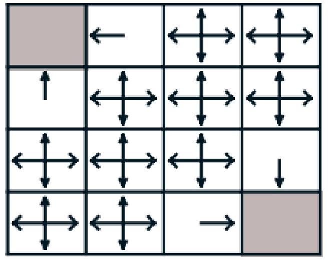
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0



← random policy

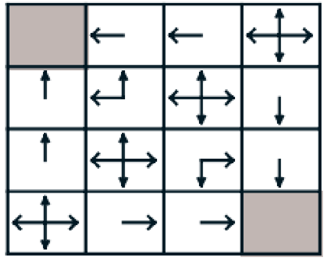
$k = 1$

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



Iterative Policy Evaluation on Small Gridworld (I)

$k = 3$

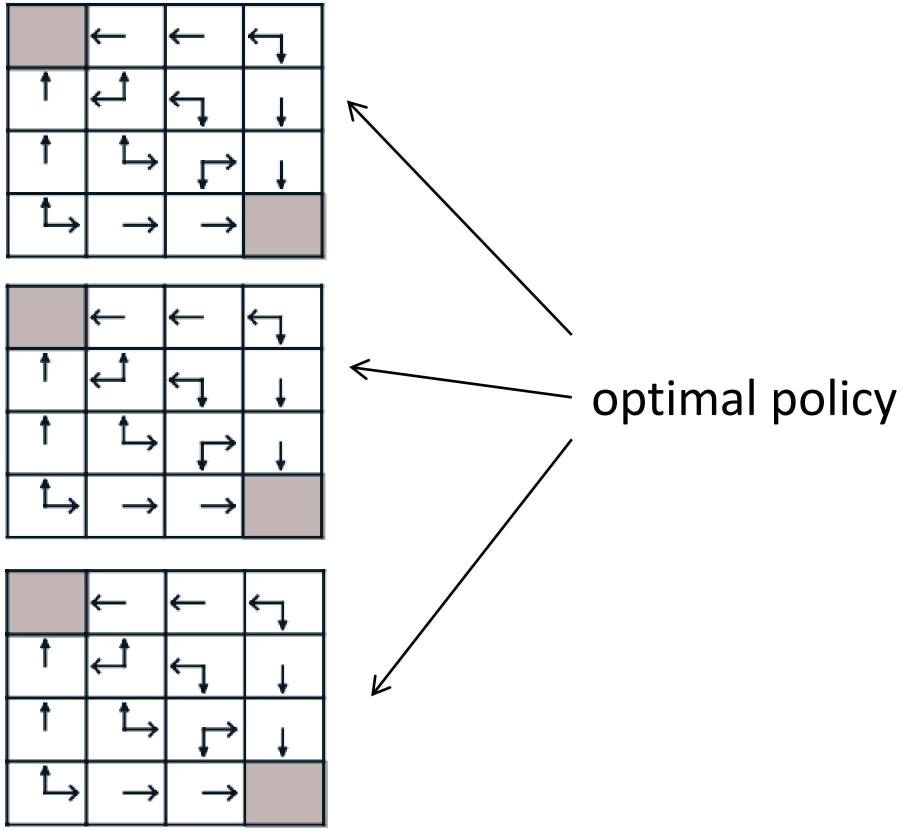
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

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

How to Improve a Policy

- ✓Given policy π
  - ✓Evaluate the policy π

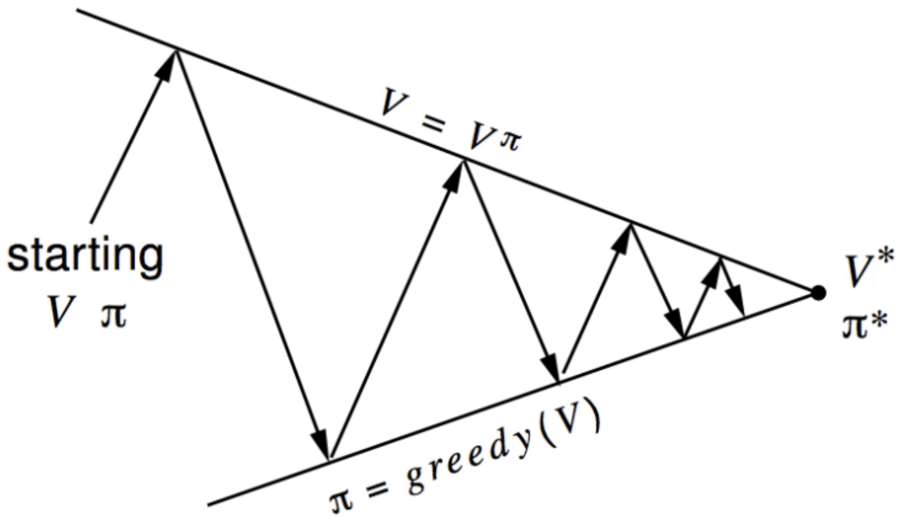
$$v_\pi(s) = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \cdots \mid S_t = s\right]$$

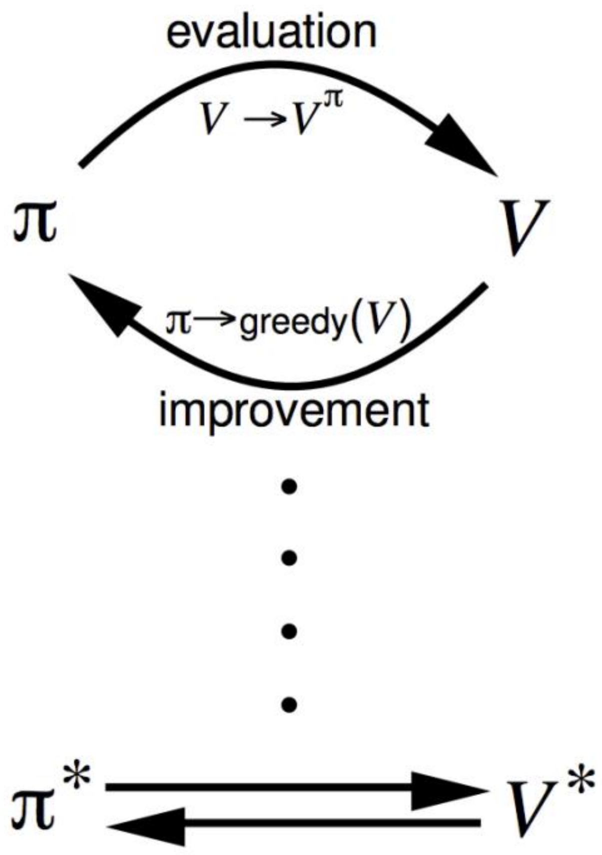
  - ✓Improve the policy by acting greedily with respect to v_π

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

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

Policy Iteration

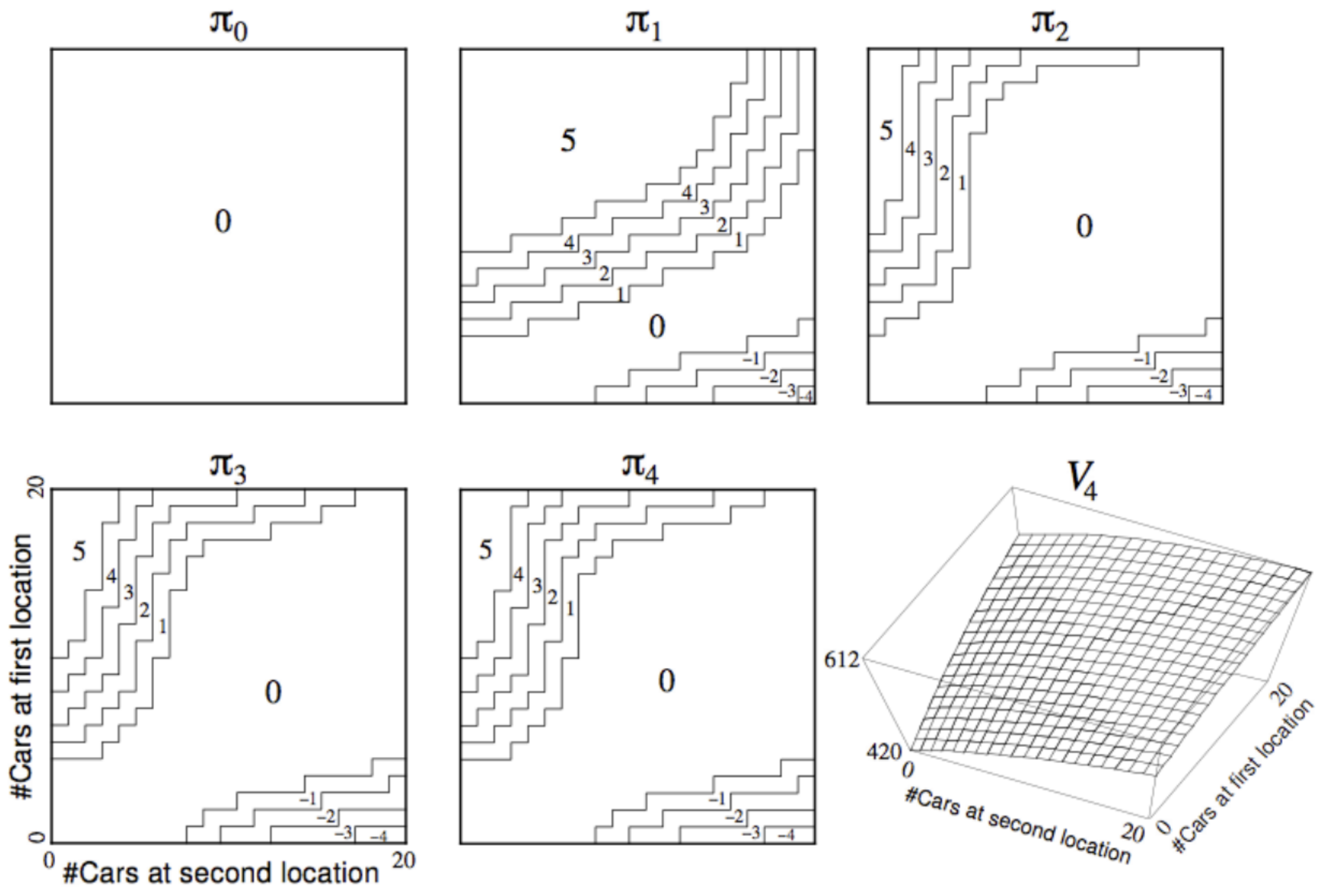




- Policy evaluation - Estimate v_π
  - Iterative policy evaluation
- Policy improvement - Generate $\pi' \geq \pi$
  - Greedy policy improvement

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



Policy Iteration in Jack's Car Rental

Policy Improvement (I)

Consider a deterministic policy $\mathrm{a} = \pi(s)$

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

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

Therefore improving the value function $v_{\pi'}(s) \geq v_\pi(s)$

Policy Improvement (II)

If improvement stops

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

We satisfy Bellman optimality

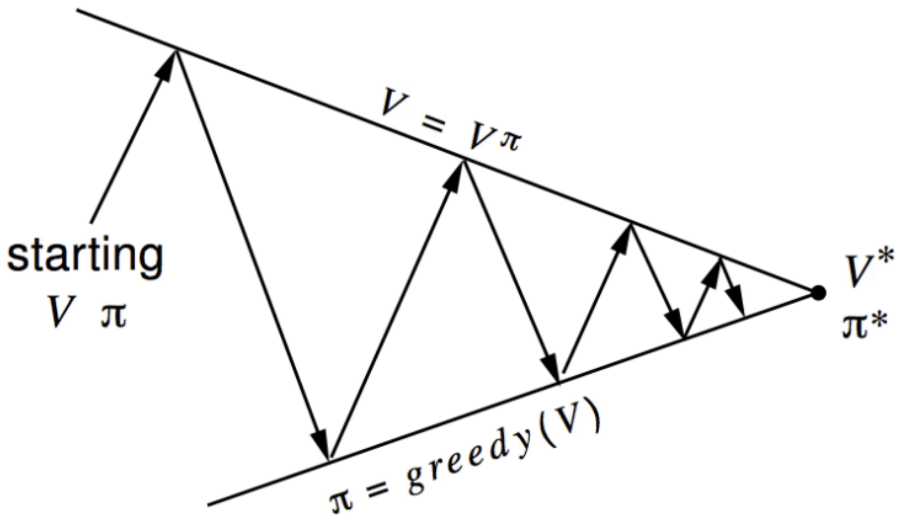
$$v_\pi(s) = \max_{a \in \mathcal{A}} q_\pi(s, a)$$

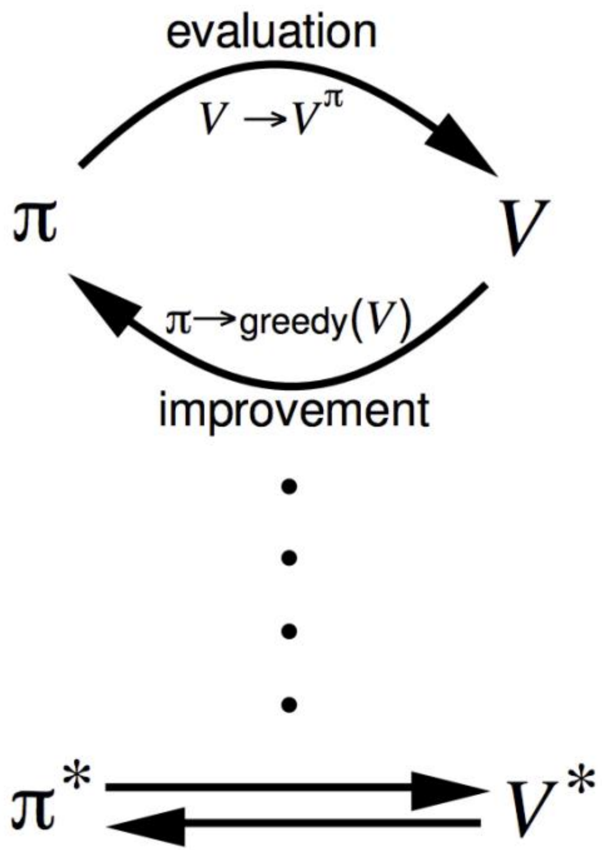
Therefore $v_\pi(s) = v_*(s), \forall s \in \mathcal{S}$, and π is an optimal policy

Modified Policy Improvement

- Does policy evaluation need to converge to v_π?
  - Introduce a stopping condition, e.g. ϵ-convergence of value function
  - Stop after k iterations of iterative policy evaluation, e.g. k=3 was sufficient in small gridworld
- Why update policy every iteration?
  - Stop after k = 1
  - This is equivalent to value iteration (coming up)

Generalized Policy Iteration





- ✓Policy evaluation - Estimate v_{π}
  - ✓Any policy evaluation
- ✓Policy improvement - Generate $\pi' \geq \pi$
  - ✓Any policy improvement algorithm

Value Iteration

Optimality Principle

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' (i.e. $v_\pi(s) = v_*(\mathrm{s})$) 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

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

- ✓Value iteration applies these updates iteratively
- ✓Intuition: start with final rewards and work backwards
  - ✓Still works with loopy, stochastic MDPs

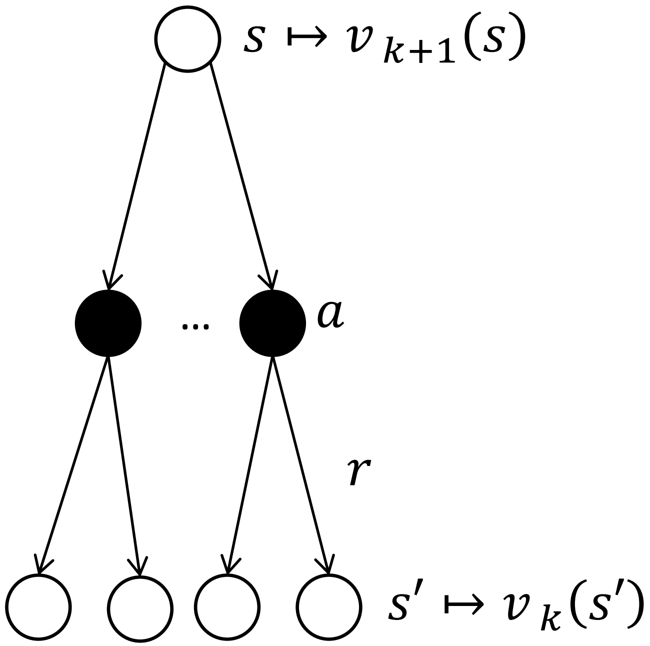
Value Iteration

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

$$v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_\pi$$

- ✓ Using synchronous backups
  i. At each iteration $k+1$
  ii. For all states $s \in \mathcal{S}$
  iii. Update $v_{k+1}(s)$ from $v_k(s')$
- ✓ Unlike policy iteration, there is no explicit policy
- ✓ Intermediate value functions may not correspond to any policy

Value Iteration - Formally



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

$$v_{k+1} = \max_{a \in \mathcal{A}} (\boldsymbol{\mathcal{R}}^a + \gamma \boldsymbol{P}^a v_k)$$

DP Example

https://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_dp.html

Synchronous Dynamic Programming Wrap-up

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 is $O(mn^2)$ per iteration ($m = |\mathcal{A}|$ and $n = |\mathcal{S}|$)
- ✓ Could also apply to action-value function $q_\pi(s,a)$ or $q_*(s,a)$
  - ✓ Complexity is $O(m^2n^2)$) per iteration

Extensions

Asynchronous Backups

- DP methods described so far used synchronous backups
  - All states are backed up in parallel
- Asynchronous DP backs up states individually, in any order
  - For each selected state, apply the appropriate backup
  - Can significantly reduce computation
  - Guaranteed to converge if all states continue to be selected

Asynchronous DP

- ✓Three simple approaches for asynchronous dynamic programming:
  - ✓In-place dynamic programming
  - ✓Prioritised sweeping
  - ✓Real-time dynamic programming

In-place dynamic programming

Synchronous value iteration stores two copies of value function

For all $s \in \mathcal{S}$

$$v_{new}(s) \leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_{old}(s')$$

$$v_{old}(s) \leftarrow v_{new}(s)$$

In-place value iteration only stores one copy of value function

For all $s \in \mathcal{S}$

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

Prioritised sweeping

- ✓ Use magnitude of Bellman error to guide state selection

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

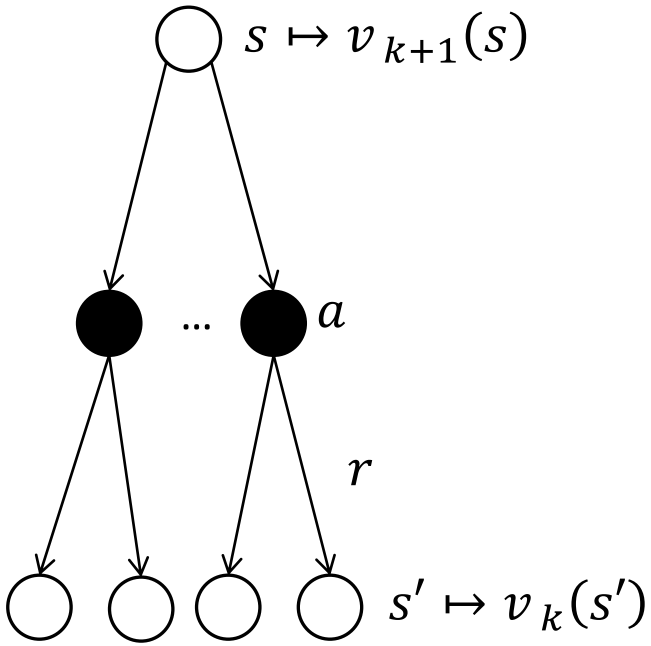
- ✓ Backup the state with the largest remaining Bellman error
- ✓ Update Bellman error of affected states after each backup
- ✓ Requires knowledge of reverse dynamics (predecessor states)
- ✓ Can be implemented efficiently by maintaining a priority queue

Real-time dynamic programming

- ✓ Intuition - 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

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

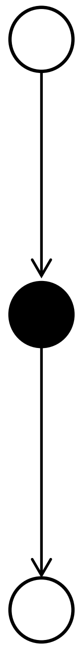
Full-Width Backup



- ✓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 Backup

- From now onwards we consider sample backups
  - Using sample rewards and sample transitions $\langle S, A, R, S' \rangle$
  - Instead of reward function $\mathcal{R}$ and transition function P
- Pros
  - Model-free - no advance knowledge of MDP required
  - Breaks the curse of dimensionality through sampling
  - Cost of backup is constant, independent of $n = |\mathcal{S}|$



Approximate Dynamic Programming

- ✓Approximate the value function
  - ✓Using a function approximator $\hat{v}(s; \boldsymbol{w})$
  - ✓Apply dynamic programming to $\hat{v}(\cdot\,; \boldsymbol{w})$
- ✓Fitted Value Iteration - For each iteration *k*
  - ✓Sample states $\tilde{\mathcal{S}} \subseteq \mathcal{S}$
  - ✓For each state s ∈ $\tilde{\mathcal{S}}$ estimate target value using Bellman optimality equation

$$\hat{v}_k(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a \hat{v}(s'; \boldsymbol{w}_k) \right)$$

  - ✓Train next value function $\hat{v}(\cdot\,; \boldsymbol{w}_{k+1})$ using targets $\{(s, \hat{v}_k(s))\}$

Wrap-up

Take (stay) home messages

- ✓ Dynamic Programming - Method for solving complex problems by breaking them down into subproblems
  - ✓ Use recursive formulation founded in return nested definition
- ✓ Policy iteration - Re-define the policy at each step and compute the value according to this new policy until the policy converges
- ✓ Value iteration - Computes the optimal state value function by iteratively improving the estimate of V(s)
- ✓ Policy vs Value iteration
  - ✓ Policy can converge quicker (agent is interested in optimal policy)
  - ✓ Value iteration is computationally cheaper (per iteration)

Next Lecture

Model-Free Prediction

- ✓Estimate the value function of an unknown MDP
- ✓Monte-Carlo approaches
- ✓Temporal-Difference learning
- ✓TD(λ)