Lecture 3: Planning by Dynamic Programming

1. Introduction

What is Dynamic Programming?

Dynamic sequential or temporal component to the problem

Programming optimising 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

Planning by Dynamic Programming

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

Other Applications of Dynamic Programming

Dynamic programming is used to solve many other problems, e.g.

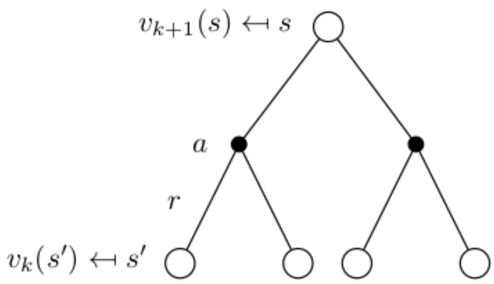
- Scheduling algorithms
- String algorithms (e.g. sequence alignment)
- Graph algorithms (e.g. shortest path algorithms)
- Graphical models (e.g. Viterbi algorithm)
- Bioinformatics (e.g. lattice models)

2. 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,
  - At each iteration $k+1$
  - For all states $s \in S$
  - 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

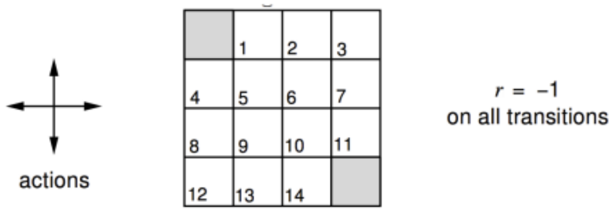
Iterative Policy Evaluation (2)



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

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

Evaluating a Random Policy in the Small Gridworld



- Undiscounted episodic MDP ($\gamma = 1$)
- Nonterminal states $1, \dots, 14$
- 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

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

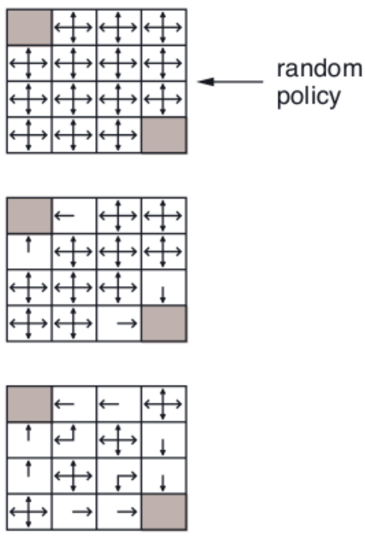
$k = 1$

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

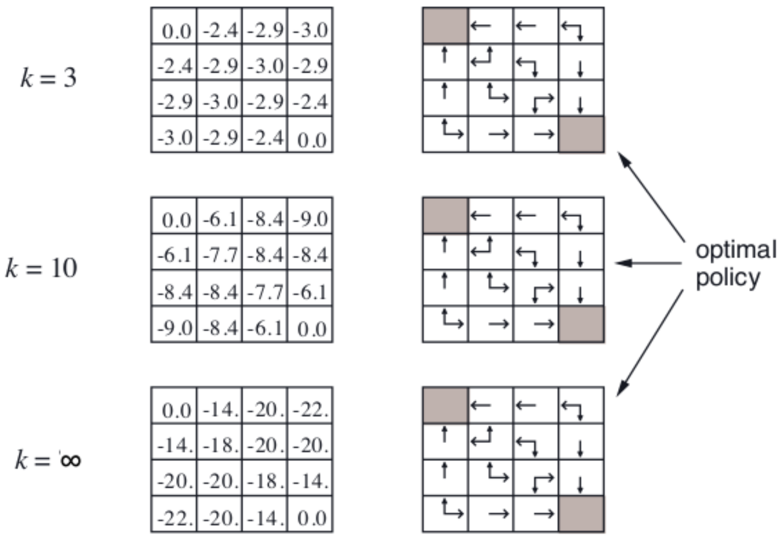
$k = 2$

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

Greedy Policy w.r.t. v_k



Iterative Policy Evaluation in Small Gridworld (2)



3. Policy Iteration

How to Improve a Policy

- Given a policy π
  - **Evaluation** the policy π

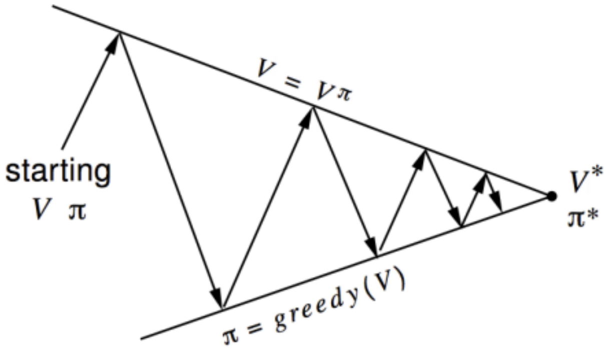
$$v_\pi(s) = E[R_{t+1} + \gamma R_{t+2} + \ldots | S_t = s]$$

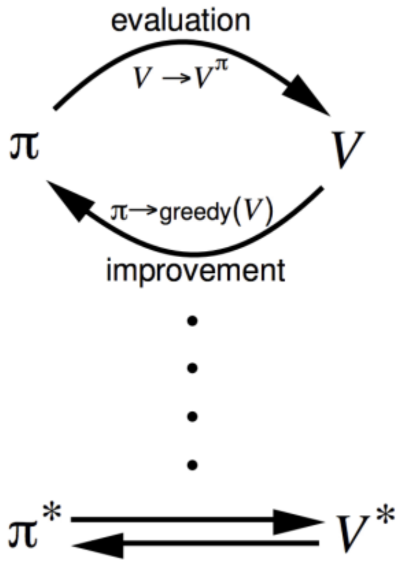
  - Improve the policy by acting greedily with respect to v_π

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

- 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





Policy evaluation Estimate v_π

Iterative policy evaluation

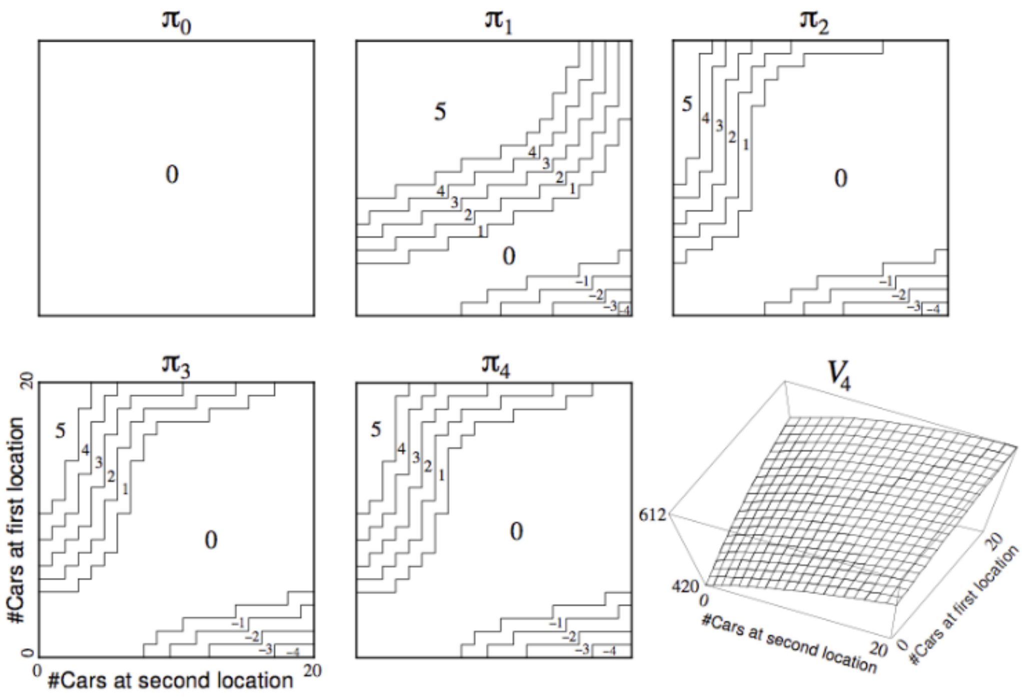
Policy improvement Generate $\pi' \geq \pi$

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 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



Policy Improvement

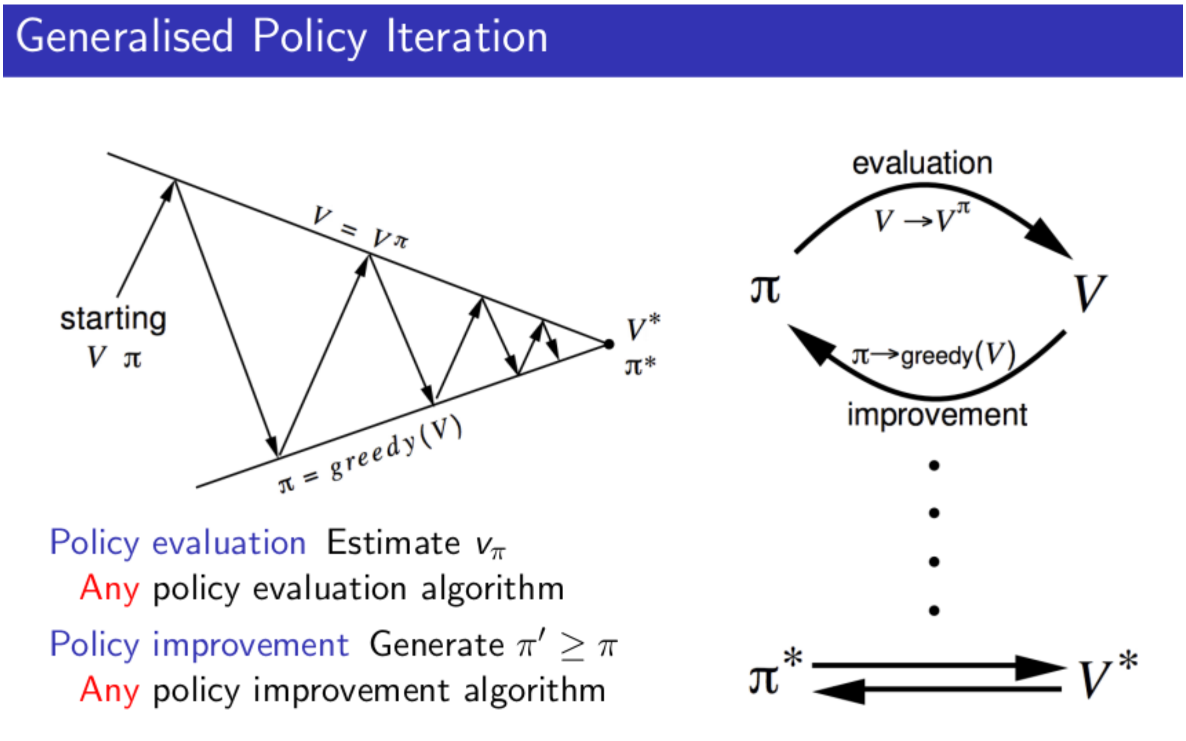
- Consider a deterministic policy, $a = \pi(s)$
- We can improve the policy by acting greedily
  - $$\pi'(s) = \arg\max_{a \in A} q_\pi(s, a)$$
- This improves the value from any state s over one step,
  - $$q_\pi(s, \pi'(s)) = \max_{a \in A} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s)$$
- It therefore improves the value function, $v_{\pi'} \geq v_\pi(s)$
  - $$\begin{aligned} v_\pi(s) &\leq q_\pi(s, \pi'(s)) = E_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\ &\leq E_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s] \\ &\leq E_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, \pi'(S_{t+2})) | S_t = s] \\ &\leq E_{\pi'}[R_{t+1} + \gamma R_{t+2} + \ldots | S_t = s] = v_{\pi'}(s) \end{aligned}$$
- If improvements stop
  - $$q_\pi(s, \pi'(s)) = \max_{a \in A} q_\pi(s, a) = q_\pi(s, \pi(s)) = v_\pi(s)$$
- Then the Bellman optimality equation has been satisfied
  - $$v_\pi(s) = \max_{a \in A} q_\pi(s, a)$$
- Therefore $v_\pi(s) = v_*(s)$ for all $s \in S$
- so π is an optimal policy

Extensions to Policy Iteration

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)



4. Value Iteration

Value Iteration in MDPs

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'

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
    - $$v_*(s) \leftarrow \max_{a \in A} R^a_S + \gamma \sum_{s' \in S} P^a_{ss'} v_*(s')$$
- 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

g			

Problem

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

V_1

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

V_2

0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-2
-2	-2	-2	-2

V_3

0	-1	-2	-3
-1	-2	-3	-3
-2	-3	-3	-3
-3	-3	-3	-3

V_4

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-4
-3	-4	-4	-4

V_5

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-5

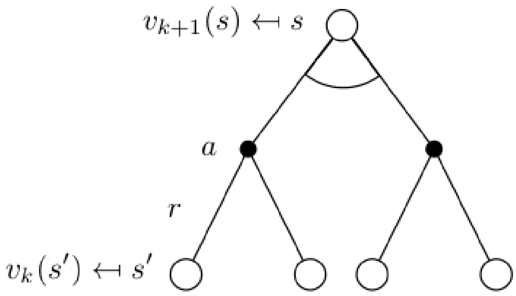
V_6

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

V_7

Value Iteration

- Problem: find optimal policy π
- Solution: iterative application of Bellman optimality backup
- $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_*$
- Using synchronous backups
  - At each iteration $k+1$
  - For all states $s \in S$
  - 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



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

Summary of DP Algorithms

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

5. Extensions to Dynamic Programming

Asynchronous Dynamic Programming

- DP methods described so far used synchronous backups
- i.e. 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

Three simple ideas 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 S
  - $$v_{new}(s) \leftarrow \max_{a \in A}(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_{old}(s'))$$
  - $$v_{old} \leftarrow v_{new}$$
- In-place value iteration only stores one copy of value function
  - for all s in S
  - $$v(s) \leftarrow \max_{a \in A}(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v(s'))$$

Prioritised Sweeping

- Use magnitude of Bellman error to guide state selection, e.g.
  - $$|\max_{a \in A}(R_s^a) + \gamma \sum_{s' \in S} P_{ss'}^a v(s') - v(s)|$$
- 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

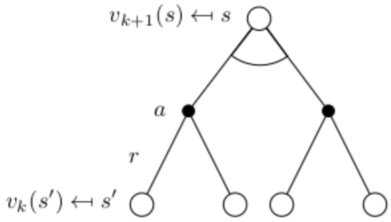
Real-Time Dynamic Programming

- 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
  - $$v(S_t) \leftarrow \max_{a \in A}(R_{S_t}^a + \gamma \sum_{s' \in S} P_{S_t s'}^a v(s'))$$

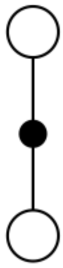
Full-width and sample backups

Full-Width Backups

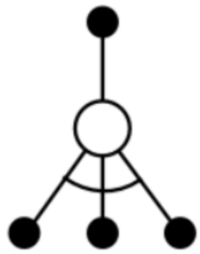
- DP uses *full-width* backups
- For each backup (sync or async)
  - Every successor state and action is considered
  - Using knowledge of the MDP transitions and reward function
- DP is effective for medium-sized problems (millions of states)
- For large problems DP suffers Bellman's *curse of dimensionality*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



Sample Backups



- In subsequent lectures we will consider *sample backups*
- 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 $n = |\mathcal{S}|$



Approximate Dynamic Programming

- 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 $\tilde{S} \subseteq S$
  - For each state $s \in \tilde{S}$, estimate target value using Bellman optimality equation,
  - $$\tilde{v}_k(s) = \max_{a \in A}(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \hat{v}(s', w_k))$$

- Train next value function $\hat{v}(\cdot, w_{k+1})$ using targets $\{< s, \tilde{v}_k(s) >\}$

6. Contraction Mapping

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 V over value functions
- There are $|S|$ 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

Value Function ∞-Norm

- We will measure distance between state-value functions u and v by the ∞-norm
- i.e. the largest difference between state values,
  - $$||u - v||_\infty = \max_{s \in S} |u(s) - v(s)|$$

Bellman Expectation Backup is a Contraction

- Define the Bellman expectation backup operator T^π,
  - $$T^\pi(v) = R^\pi + \gamma P^\pi v$$
- This operator is a γ-constraction, i.e. it makes value functions closer by at least γ,
  - $$\begin{aligned} ||T^\pi(u) - T^\pi(v)||_\infty &= ||(R^\pi + \gamma P^\pi u) - (R^\pi + \gamma P^\pi v)||_\infty \\ &= ||\gamma P^\pi (u - v)||_\infty \\ &\le ||\gamma P^\pi ||u - v||_\infty||_\infty \\ &\le \gamma||u - v||_\infty \end{aligned}$$

Contraction Mapping Theorem

For any metric space V that is complete (i.e. closed) under an operator $T(v)$, where T is a γ-contraction,

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

Convergence of Iter. Policy Evaluation and Policy Iteration

- 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

- Policy iteration converges on v_*

Bellman Optimality Backup is a Contraction

- Define the Bellman optimality backup operator T^*,
- $$T^*(v) = \max_{a \in A} R^a + \gamma P^a v$$
- This operator is a γ-contraction, i.e. it makes value functions closer by at least γ (similar to previous proof)
- $$||T^*(u) - T^*(v)||_\infty \leq \gamma ||u - v||_\infty$$

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_*