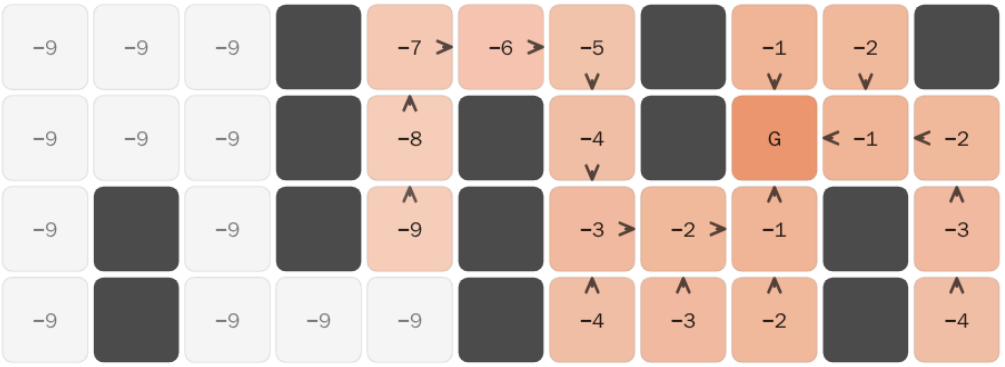
Planning with a Model



Reinforcement learning, part 2: dynamic programming, policy iteration and value iteration

AUTHOR

Etienne Chassaing[1, 2]

AFFILIATIONS

1. Hugging Face
2. CentraleSupélec

PUBLISHED

Oct. 2026

Table of Contents

1 Introduction

- 1.1 Where Part 1 left us

2 1. Evaluating a policy via dynamic programming

- 2.1 1.1 What dynamic programming is
- 2.2 1.2 Iterative policy evaluation

3 2. Policy iteration: evaluate, improve, repeat

- 3.1 2.1 Improving a policy
- 3.2 2.2 Policy iteration
- 3.3 2.3 A larger example: Jack's car rental

4 3. Value iteration: optimize directly

- 4.1 3.1 One sweep of the optimality backup
- 4.2 3.2 Why does it converge?

5 4. Taking a step back

- 5.1 4.1 Evaluate and improve: generalized policy iteration
- 5.2 4.2 One optimal value function, many optimal policies
- 5.3 4.3 Beyond full sweeps

6 5. Wrap-up

7 Extra exercises

Introduction

In the previous part, we laid the foundations of reinforcement learning: you now have the tools to describe a decision problem and to say what an optimal solution to it is.

This second part is about planning with a model: evaluating a policy, or finding the best one, in an environment with rules we know. This applies to problems such as video games, chess or Go, and it is a crucial step before learning from experience, without a model, which is the real goal of reinforcement learning.

Where Part 1 left us

Part 1, *Modeling the World* gave us the basics: an agent in state S_t takes an action A_t, receives a reward R_{t+1} and lands in S_{t+1}, according to the dynamics of the Markov Decision Process (MDP), $p(s', r \mid s, a)$.

The agent wants to maximize the return, recursively defined by the relation $G_t = R_{t+1} + \gamma\, G_{t+1}$, or in full, as the discounted sum of all the rewards to come:

$$G_t = R_{t+1} + \gamma\, R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

To evaluate how good a state or a state-action pair is, we defined helper functions: the value functions score states and actions under a policy π,

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s], \qquad q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a],$$

and their optimal counterparts $v_*(s) = \max_\pi v_\pi(s)$ and $q_*(s, a) = \max_\pi q_\pi(s, a)$. The state values satisfy the Bellman equations, one per state:

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

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

On the four-state altitude map, Part 1 solved these by hand, as a system of equations.

This page teaches you how to solve them on any finite MDP, with algorithms that only need the model p and the recursive structure of these equations. We will study:

1. Dynamic programming and policy evaluation: computing the value of a policy.
2. Policy iteration: finding the best policy in three steps: evaluate, improve, repeat.
3. Value iteration: a shortcut to the optimal behavior: optimize directly, one sweep at a time.
4. Taking a step back: the pattern behind all three, generalized policy iteration, and why the optimal policy is not unique.

Part 3 will remove the model: Part 3.1, *Learning from Experience*, with Monte Carlo methods and temporal-difference learning, then Part 3.2, *Learning to Act*, with SARSA and Q-learning, learn the same values from sampled interactions only. Everything they do is an approximation at scale of what this page computes exactly.

1. Evaluating a policy via dynamic programming

To be able to find an optimal policy, we first need to be able to compare policies. Policy evaluation is the computation of the value function of a policy, in every state. When a model is available, the main technique to do so is to use dynamic programming.

1.1 What dynamic programming is

Dynamic programming (DP) solves a complex problem by breaking it into simpler subproblems, storing their solutions, and reusing them instead of computing them again. Here, "dynamic" refers to the sequential, temporal side of the problem. It applies when a problem has two properties:

- optimal substructure: an optimal solution is made of optimal solutions of subproblems;
- overlapping subproblems: the same subproblems come back many times, so their solutions are worth caching.

MDPs have both. The Bellman equation is the decomposition link: the value of a state is one step plus the value of the next state. And the value function is the cache: one number per state, computed once, reused by every state that leads there.

The first property has a name in this setting, the principle of optimality:

> PRINCIPLE OF OPTIMALITY
> *A policy is optimal from a state s if and only if it is optimal from every state s' it can reach from s.*

Once the hiker of Part 1 has reached A, whatever happened before, the best the hiker can do is to behave optimally from A. The problem splits into a first step and a smaller problem from the next state, which is exactly the Bellman optimality equation.

DP assumes full knowledge of the MDP: the states, the actions, the dynamics p and γ. It is a planning method, used for both problems of section 4.3 of Part 1:

- Prediction/evaluation: given the MDP and a policy π, compute v_π;
- Control: given the MDP, compute v_* and an optimal policy π_*.

1.2 Iterative policy evaluation

The Bellman expectation equation gives $|\mathcal{S}|$ linear equations for the $|\mathcal{S}|$ values $v_\pi(s)$. We could solve the system directly, but that costs about $|\mathcal{S}|^3$ operations, too much for large state spaces.

Instead, we turn the equation into an update rule, and apply it again and again. Starting from any v_0, for instance 0 everywhere, and computing, for every state,

$$v_{k+1}(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma\, v_k(s') \right]$$

One pass over all the states is a sweep. Each sweep replaces the value of every state by a one-step look-ahead on the current values, a *Bellman backup*, and the sequence $v_0, v_1, v_2, \ldots$ converges to v_π. In practice, we stop when the largest change of a sweep, Δ, falls below a small threshold θ:

Algorithm Iterative policy evaluation

Input: a policy π, a threshold $\theta > 0$

$v(s) \leftarrow 0$ for every state s
repeat
  $\Delta \leftarrow 0$
  for each state s
    $v_{\text{old}} \leftarrow v(s)$
    $v(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma\, v(s')\big]$
    $\Delta \leftarrow \max\big(\Delta, |v(s) - v_{\text{old}}|\big)$
until $\Delta < \theta$

One sweep costs about $|\mathcal{S}|^2|\mathcal{A}|$ operations: every state, every action, every next state, but we don't know how many sweeps are required to converge.

Main example of this lesson: A 4×4 grid with states 0 to 15, numbered row by row; the corners 0 and 15 are terminal. Four actions, up, down, left, right, deterministic; an action that would leave the grid leaves the agent in place. Every step costs $r = -1$, $\gamma = 1$, and the policy we study is the equiprobable random one, $\pi(a \mid s) = \frac{1}{4}$.

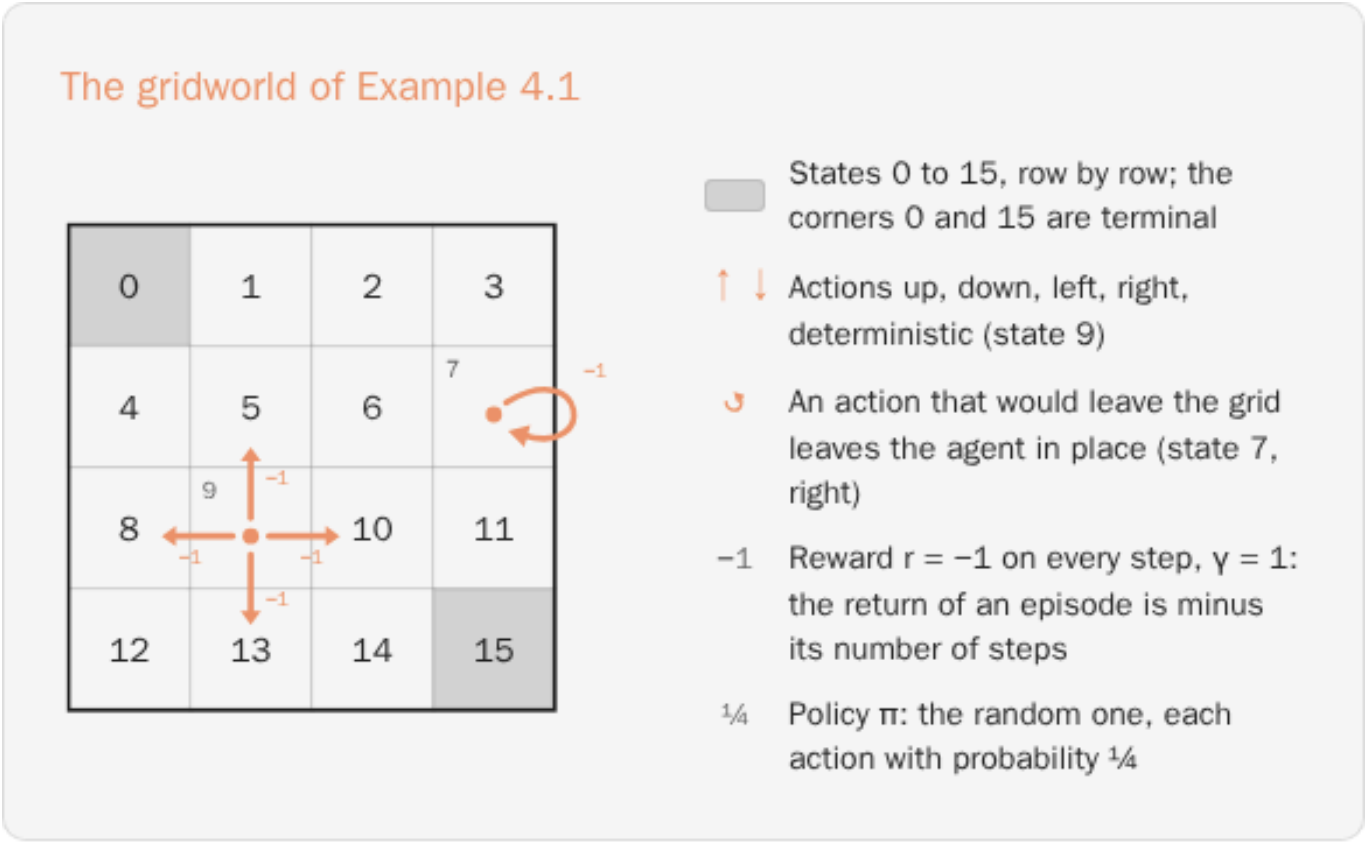


Figure 1 · Grey: the two terminal corners. The arrows show the four actions from state 9, and, from state 7, a move right that would leave the grid and leaves the agent in place.

Starting the policy evaluation from $v_0 = 0$: after one sweep, every non-terminal state is worth -1, the reward of one step, whatever happens later. For state 1, next to the terminal corner:

$$v_1(1) = \tfrac{1}{4}\big[(-1 + v_0(1)) + (-1 + v_0(5)) + (-1 + v_0(0)) + (-1 + v_0(2))\big] = -1$$

After two sweeps, the states next to a terminal corner are worth -1.75 (one of their four moves ends the episode) and the others -2. The values keep spreading from the corners, sweep after sweep, up to convergence:

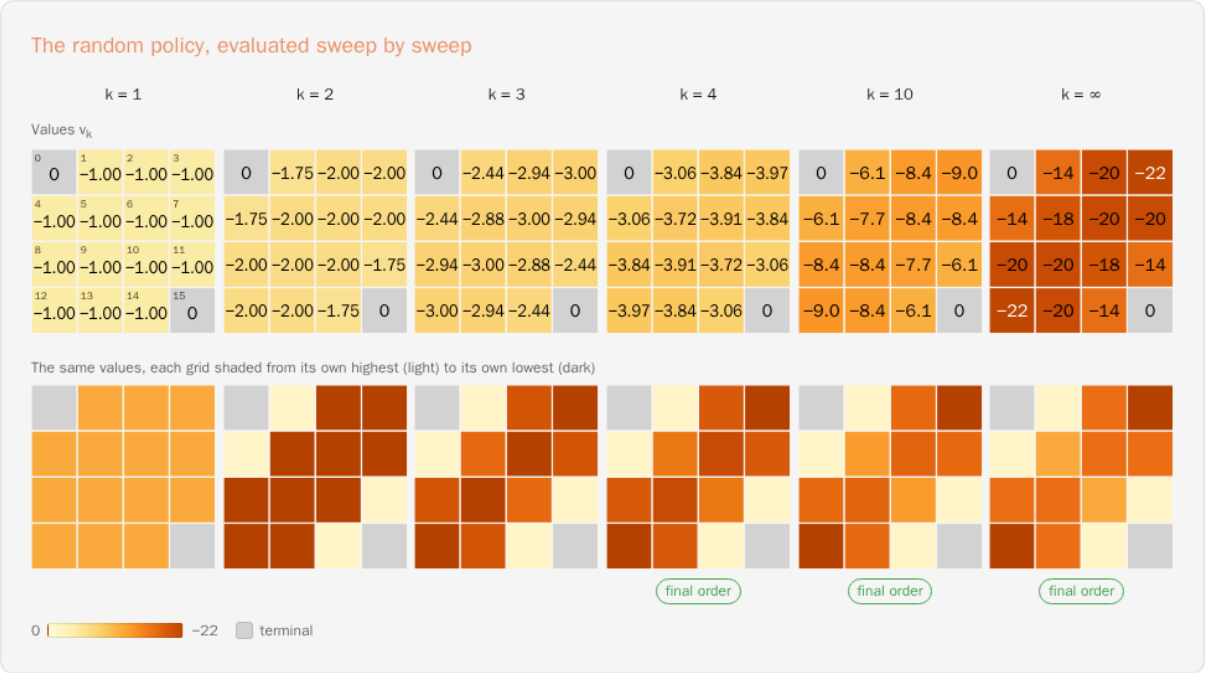


Figure 2 · Top: the values v_k of the random policy after k sweeps (two arrays), on one color scale for every grid, darker when more negative; the grey corners are terminal. Bottom: the same values, each grid shaded against itself, from its highest to its lowest value. The numbers keep falling up to k = ∞, but from k = 4 the bottom grids barely change: the order between the states is final.

To what values does $v_\pi(s)$ converge? Take a few seconds to think about it.

ANSWER Open once you have a guess ›

The values take a long time to fully settle: about 220 sweeps to get within 10^{-4} of the true v_π, the far corners going from -4 after 4 sweeps to -22. However, note that the comparisons between states settle much faster: from $k = 4$, every state that is better than another under v_π is already better under v_k. Keep this in mind for sections 2 and 3.

These large negative values say little about the task: they measure the cost of acting at random. Finding the optimal cost is the job of the next step: improving the policy.

2. Policy iteration: evaluate, improve, repeat

Evaluating a policy is a great step forward, but our true goal is to find the best policy, or one of the best: this is what policy iteration is about.

2.1 Improving a policy

Once we know v_π, we can measure what happens if, in one state s, we make a different choice and take a different action a than the policy would, then follow π again. That is exactly $q_\pi(s, a)$, and it follows from v_π in one step:

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

On the gridworld of section 1.2, under the random policy, moves are deterministic so the sum shrinks and $q_\pi(s,a) = -1 + v_\pi(s'_a)$, where s'_a is the cell reached by a. Moving down from state 11 ends the episode, and a terminal state is worth 0: $q_\pi(11, \text{down}) = -1 + 0 = -1$. Moving down from state 7 lands in state 11: $q_\pi(7, \text{down}) = -1 + (-14) = -15$.

Same first action, but very different values, because q_π scores one step of your choice followed by the random policy: from 11, that one step ends the episode; from 7, it hands the agent back to 14 more random steps, on average.

$$q_\pi(7, \text{down}) = \underbrace{-1}_{\substack{\text{one step,}\\ \text{your choice}}} + \underbrace{v_\pi(11)}_{\substack{\text{then } \pi \text{ from 11}\\ \text{until a corner}}} = -1 + (-14) = -15$$

The figure below starts with a simpler policy than the random one, *up, then left*, whose path is easy to follow: there, $q_\pi(7, \text{down}) = -6$. Switch it to random to find the -15 above, as the average of many episodes.

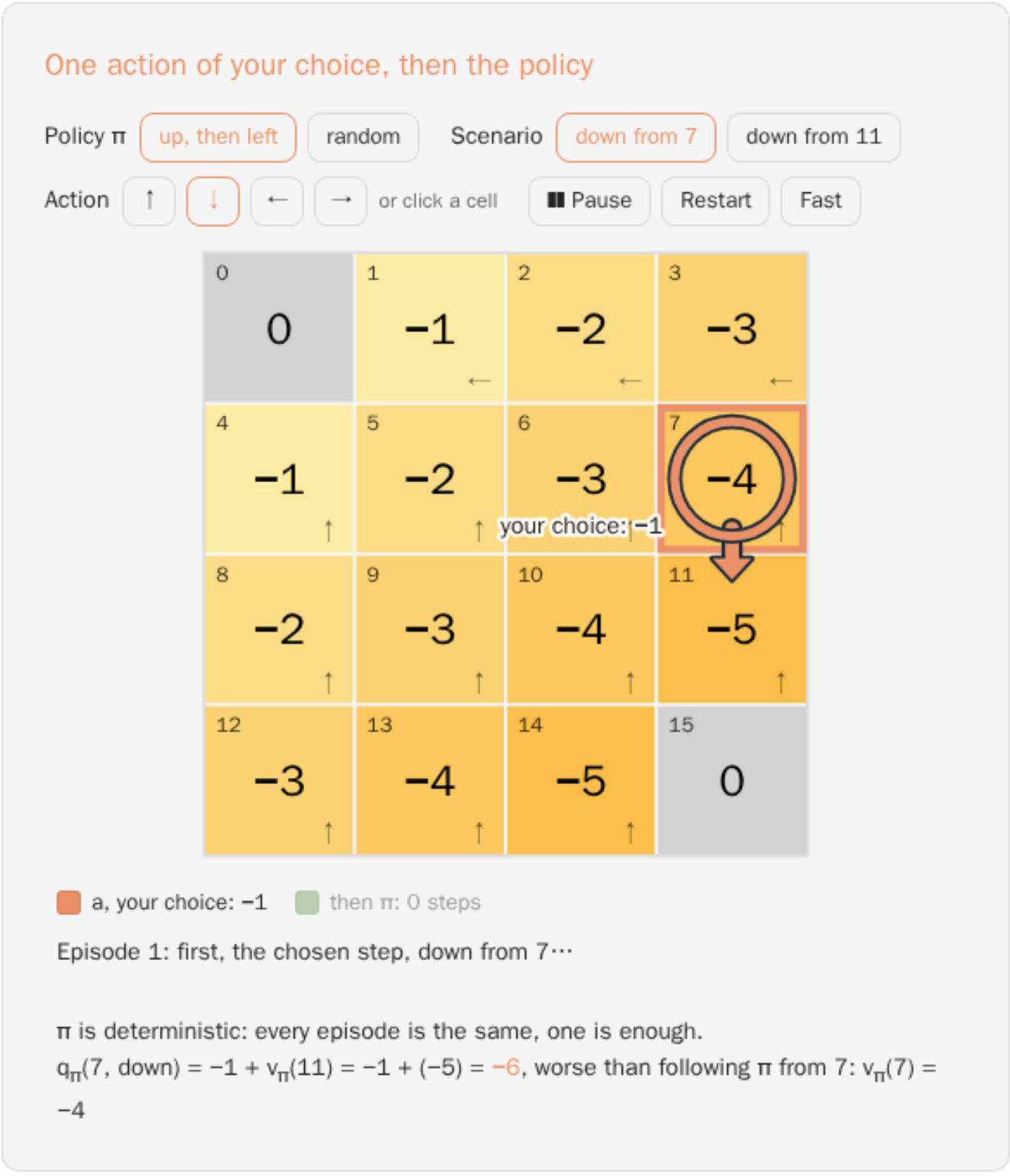


Figure 3 · The values v_π of a policy π. By default, π is a fixed policy, easy to follow: up, then left along the top row (small arrows). The orange arrow is the action a, chosen once; from there, the walker follows π (green trail) until it reaches a corner. $q_\pi(s, a)$ is −1 for the chosen step plus v_π of the cell reached. Switch π to random for the policy of the text: every episode is then different, and q_π is the average of their returns.

The figure already holds the idea of this section. Under the random policy, state 7 is worth $v_\pi(7) = -20$; going down first, then acting at random, is worth $q_\pi(7, \text{down}) = -15$.

One better choice, made once, saves five steps on average. In general, if $q_\pi(s, a) > v_\pi(s)$, taking a in s is better than following π there (both are expected returns from s). Changing the policy in that one state should help, but is this always the case, in any order we proceed? The policy improvement theorem says that it is, even when we change it in every state at once.

> POLICY IMPROVEMENT THEOREM
>
> *Let π and π' be two deterministic policies such that, for every state s,*
>
> $$q_\pi(s, \pi'(s)) \geq v_\pi(s).$$
>
> *Then π' is at least as good as π: $v_{\pi'}(s) \geq v_\pi(s)$ for every s.*

The natural candidate here is what we call the greedy policy, which picks in every state the action with the best return one step ahead:

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

It satisfies the hypothesis of the theorem by construction. In every state, $v_\pi(s)$ is an average of the action values, weighted by the policy, and a max is at least an average:

$$q_\pi(s, \pi'(s)) = \max_a q_\pi(s, a) \;\geq\; \sum_a \pi(a \mid s)\, q_\pi(s, a) = v_\pi(s)$$

So greedifying never hurts.

When does it stop helping? Only when the policy is already optimal:

OPTIONAL A stable policy is optimal: why greedifying changes nothing only at the optimum ●●○ Reason ›

Exercise 2.1 Proof of the policy improvement theorem ●●● Prove Show ›

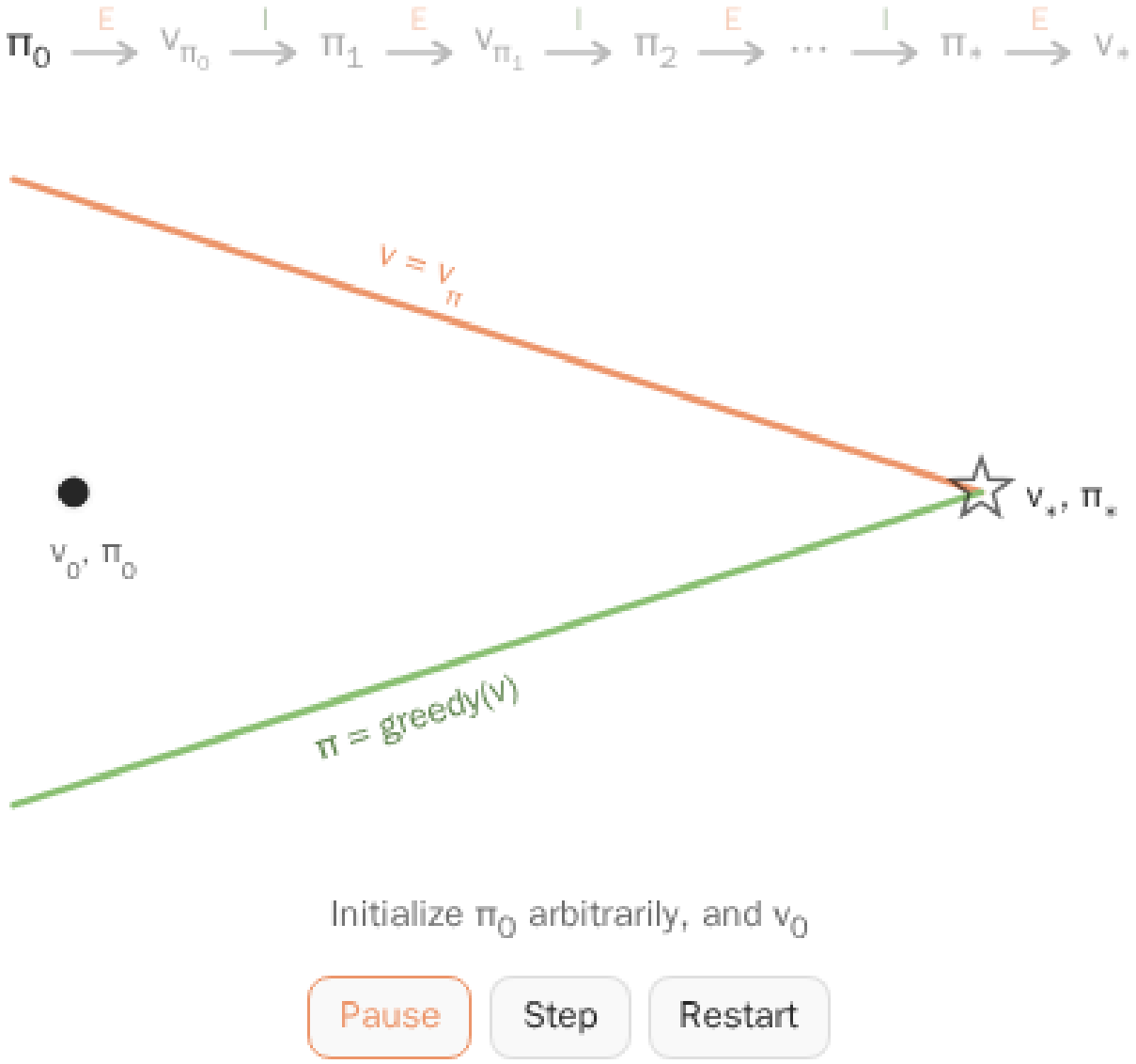
2.2 Policy iteration

Policy iteration alternates the two steps until the policy stops changing:

1. Evaluate the current policy: compute v_π, with the sweeps of section 1.2.
2. Improve it: make it greedy with respect to v_π.
3. Repeat until the policy is stable: $\pi' = \pi$.

Algorithm Policy iteration

initialize π arbitrarily
repeat
  policy evaluation
    repeat for each state s
      $v(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma\, v(s')\big]$
    until convergence
  policy improvement
    $\pi_{\text{old}} \leftarrow \pi$
    for each state s
      $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma\, v(s')\big]$
until $\pi = \pi_{\text{old}}$



Policy iteration, with the line being run highlighted, and a schematic run in two pictures. Top, the chain of evaluations (E) and improvements (I). Bottom, the space of value functions and policies: every evaluation brings the values onto the line $v = v_\pi$, sweep after sweep; every improvement brings the policy onto the line $\pi = \text{greedy}(v)$, in one step. The two lines meet at (v_*, π_*).

In words: each policy is strictly better than the previous one, unless it is already optimal. And policy iteration always stops: a finite MDP has a finite number of deterministic policies, so policy iteration reaches an optimal one in a finite number of iterations: v_*, which is unique, and an optimal policy, which may not be.

On the gridworld of section 1.2, it goes fast. The greedy policy with respect to the values of the random policy is already optimal: walk to the nearest corner. Better still, it is optimal long before the evaluation has converged: the greedy policy of v_3 already is, although v_3 is far from v_π. The greedy step only reads the ranking of the neighbors of each state, not the values themselves, nor the order of all the states: that is why it settles at $k = 3$, one sweep before the full order of section 1.2.

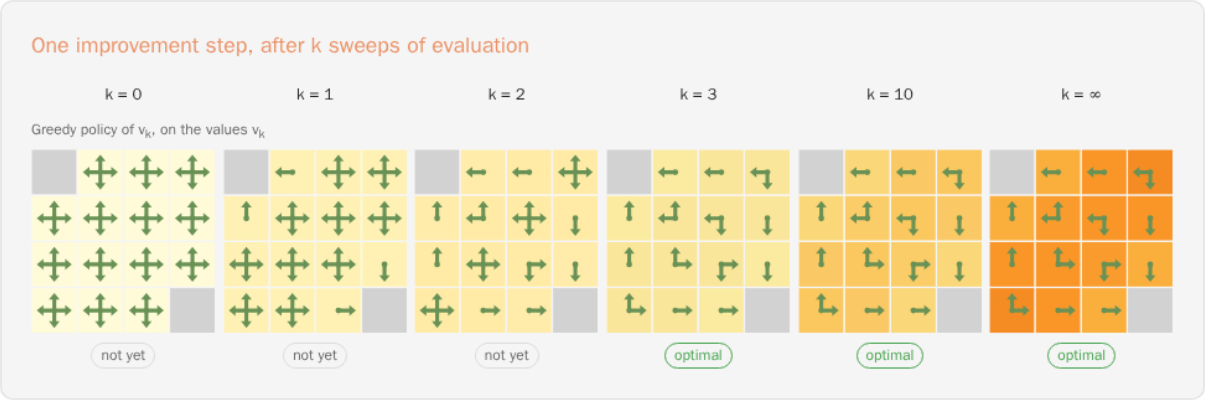


Figure 5 · The same sweeps as in section 1.2, with, on each grid, the greedy policy of v_k (green arrows, several where actions tie). At k = 0 every action ties: the greedy policy is the random one again. From k = 3, every greedy action leads toward the nearest corner: one improvement step gives an optimal policy, long before the values have converged.

Exercise 2.2 One improvement step on a 3 × 3 grid Compute Show ›

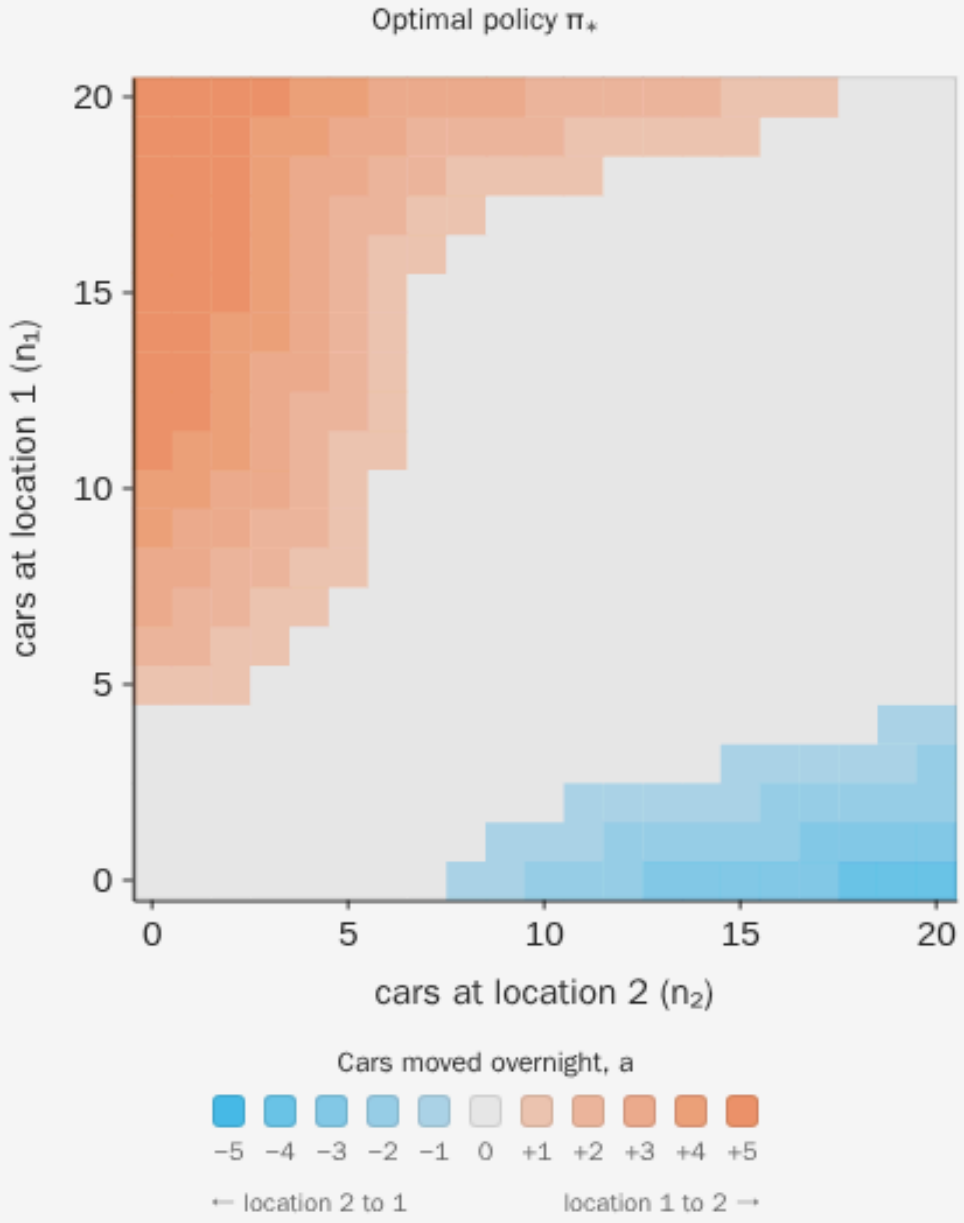
2.3 A larger example: Jack's car rental

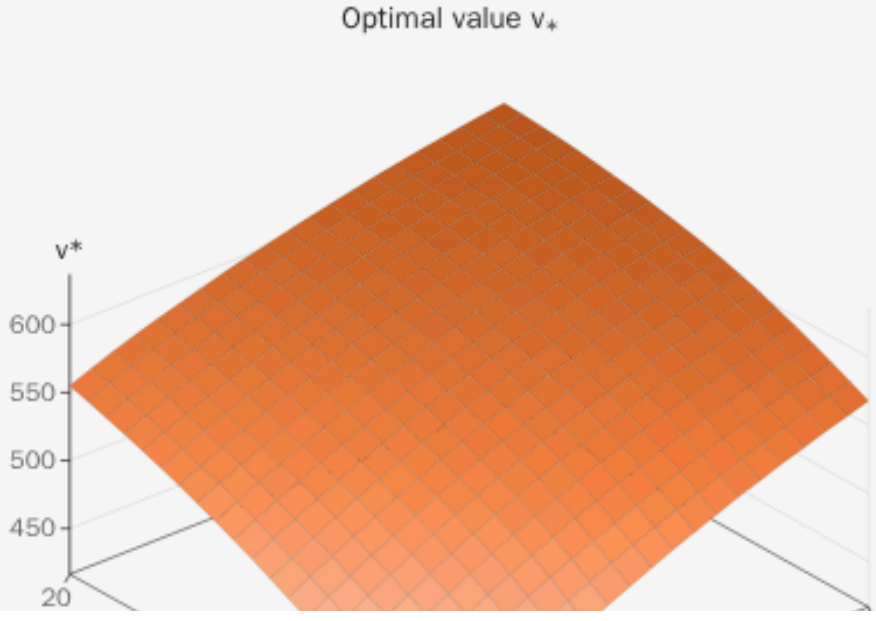
Let's take as an example a problem that is not a grid, with random dynamics, and too many states to read as a table.

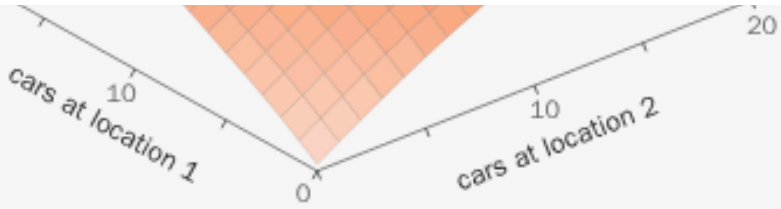
Jack manages two car rental locations. Each day, customers come to each location to rent cars, and other customers return cars, in random numbers that follow Poisson distributions with known rates (3 and 4 requests a day, 3 and 2 returns). Jack earns $10 per car rented, if he has one. Overnight, he can move up to 5 cars between the two locations, at $2 per car moved. Each location holds at most 20 cars, and $\gamma = 0.9$.

The state is the number of cars at each location, $21 \times 21 = 441$ states; the action is the number of cars moved overnight, from -5 to $+5$. The dynamics are random but fully known, so $p(s', r \mid s, a)$ can be computed exactly: this is a planning problem. Starting from the policy that never moves any car, policy iteration converges in four improvements, $\pi_4 = \pi_*$. With 441 states, tables of numbers are no longer readable: the policy becomes a map, and the value function a surface.

Jack's car rental, solved







Hover a state to find it on the other side; drag the surface to turn it. reset view

Figure 6 · The optimal policy π_*, one cell per state: its color is the number of cars moved overnight, positive from the first location to the second. Next to it, the optimal value v_* over the same 441 states; drag to turn it. More cars are worth more. The policy mostly sends cars to the second location, which rents more and gets fewer back, and moves a few the other way only when the first location is nearly empty.

Policy iteration already handles problems with hundreds of states. And with that many states, the value function looks less like a table than like a smooth surface: approximating it by a continuous function, instead of storing one number per state, is the idea of Part 4, *From Discrete to Continuous*.

INTERACTIVE SPACE Jack's car rental, policy iteration

The whole run, from π_0 to π_*: click a state to see its 11 action values, and compare full, exact and truncated evaluation (m = 1 is value iteration).

▶ Run it here Open in a new tab ↗

3. Value iteration: optimize directly

Can we go faster than policy iteration? Are there shortcuts to its cumbersome full evaluations? Value iteration is one of them.

3.1 One sweep of the optimality backup

Policy iteration evaluates each policy to convergence before improving it, and section 2.2 showed that this is often wasted work: the greedy policy settles long before the values do.

Value iteration goes to the extreme, and converges directly to the optimal values v_*. It skips the explicit policy, and applies the Bellman optimality equation as an update rule:

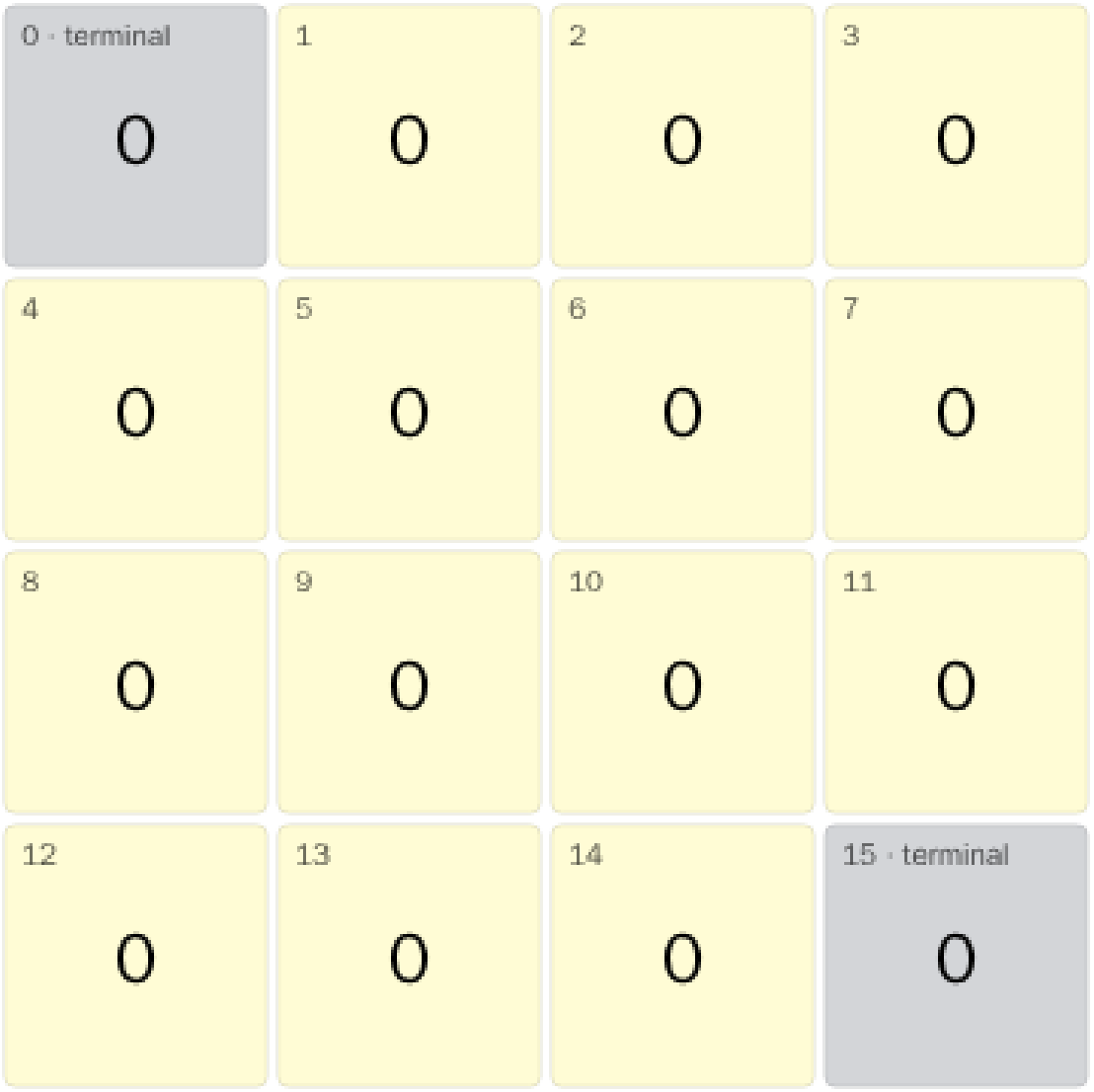
$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma\, v_k(s') \right]$$

Same sweeps as policy evaluation, with a max over the actions instead of an average under π. This has the consequence to converge to the optimal value policy (because of the properties of the Bellman equations). When the values have converged, one last greedy sweep extracts the policy:

Algorithm Value iteration

Input: a threshold $\theta > 0$

$v(s) \leftarrow 0$ for every state s
repeat
  $\Delta \leftarrow 0$
  for each state s
    $v_{\text{old}} \leftarrow v(s)$
    $v(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma\, v(s')\big]$
    $\Delta \leftarrow \max\big(\Delta, |v(s) - v_{\text{old}}|\big)$
until $\Delta < \theta$
return $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma\, v(s')\big]$



Before the first sweep Δ = –

v(s) ← 0 for every state

Pause Step End of sweep Restart

Value iteration on the gridworld of section 1.2, in place, one state at a time. The highlighted line is the one being run; on the grid, the state being updated, the neighbors its max reads, and the action that wins. The values settle one ring of cells per sweep; the fourth sweep changes nothing, $\Delta = 0$, and the greedy policy is read off.

A sweep costs the same as a sweep of policy evaluation, about $|\mathcal{S}|^2|\mathcal{A}|$ operations. Each one is an improvement *and* a partial evaluation at once. This method is often preferred to Policy Iteration, but its number of sweeps can somletimes be higher.

Exercise 3.1 Value iteration on the 3 × 3 grid ●○○ Compute Show ›

3.2 Why does it converge?

On our grids, value iteration stops after a few sweeps. With $\gamma < 1$ and random dynamics, as in Jack's car rental, v_k never reaches v_* exactly: it only gets closer at every sweep. That it always converges, and to v_* only, comes from one property of the update, folded below for those who want the math.

OPTIONAL Why do these algorithms converge?: the Bellman operator is a contraction ●●● Prove ›

4. Taking a step back

Policy iteration and value iteration look like two different algorithms. Before wrapping everything up, let's take a step back on what they share: the same two processes underneath, and the same answer at the end, a unique optimal value function, with possibly different optimal policies.

4.1 Evaluate and improve: generalized policy iteration

Side by side, the two algorithms that find π_*:

- Policy iteration: each iteration is a full evaluation followed by a greedy step. It reaches π_* in a finite number of iterations: few iterations, each one expensive.
- Value iteration: each iteration is one sweep of the optimality backup. It reaches v_* in the limit (exactly, on our grids): many sweeps, each one cheap.

Both are instances of one idea, generalized policy iteration (GPI): two processes that interact, an evaluation that makes the values consistent with the current policy, and an improvement that makes the policy greedy with respect to the current values. Together they converge to the only point that both accept, v_* and π_*. Almost every method of Part 3 is GPI with an approximate evaluation.

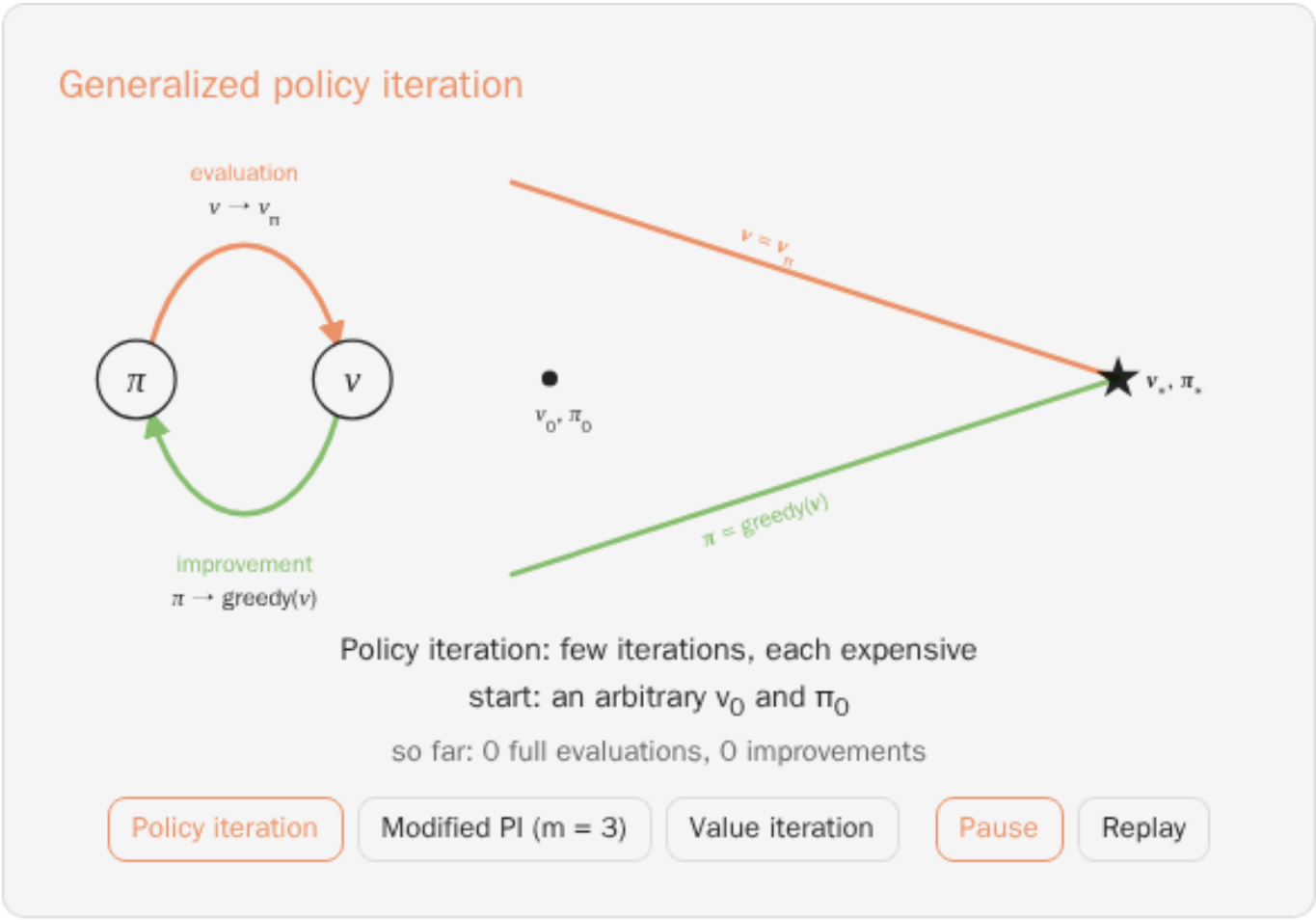


Figure 8 · Left: the two processes, evaluation and improvement, each one pulling toward its own goal. Right: the same, as two lines that meet at (v_*, π_*). Evaluation moves the values toward the line $v = v_\pi$, improvement moves the policy onto the line $\pi = \text{greedy}(v)$. Policy iteration reaches the line $v = v_\pi$ at every cycle, value iteration takes a single sweep toward it, modified policy iteration takes m.

4.2 One optimal value function, many optimal policies

Exercise 3.1 ended on a fact worth a second example: the optimal value function is unique, the optimal policy is not.

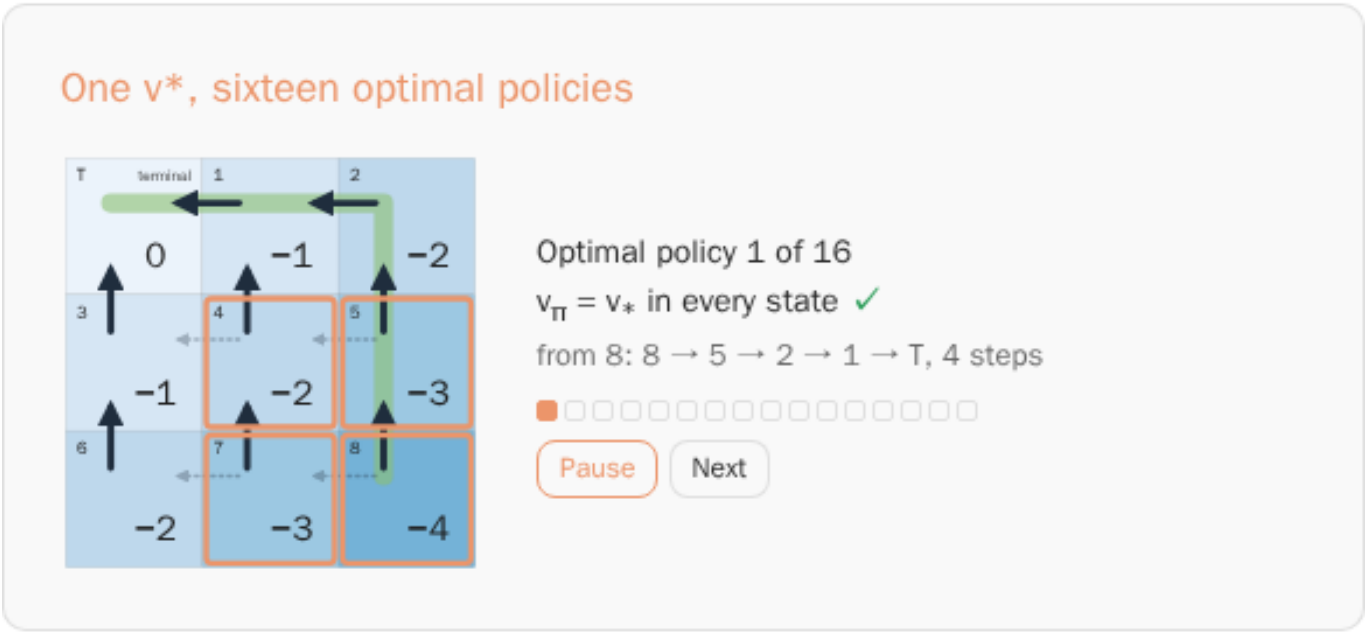


Figure 9 · The 3 × 3 grid of Exercise 3.1 and its optimal values v_*, which never change. The arrows run through the 16 deterministic optimal policies, which only differ in the four outlined states, where up and left tie; the green band is the path from 8.

In the gambler's problem of the extra exercises (E.1 and E.2), the ties are not an accident of symmetry: with a fair coin, every policy is optimal.

4.3 Beyond full sweeps

Every algorithm of this page sweeps the whole state set, again and again: fine for 16 states, hopeless for a Rubik's cube. Updating only some states, in any order, still works, and is the first step toward Part 3.

OPTIONAL Asynchronous dynamic programming: sweeps don't scale ›

5. Wrap-up

To wrap up, back to the hike of Part 1, solved with the two algorithms of this page.

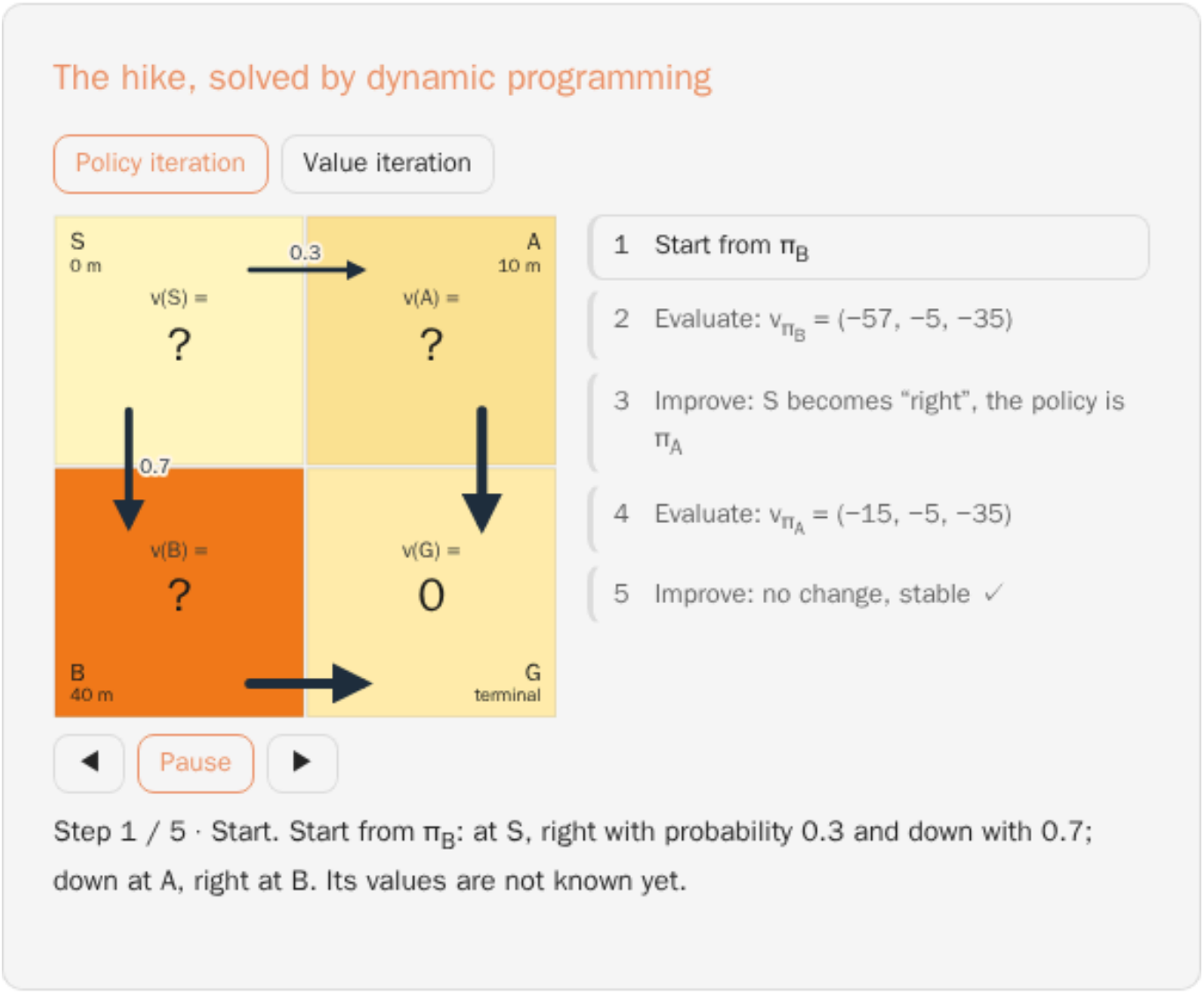


Figure 10 · The altitude map of Part 1, $\gamma = 1$. Policy iteration starts from π_B: one evaluation, one improvement, and the policy is π_A, which a second cycle confirms. Value iteration starts from $v_0 = 0$: step through the sweeps and watch the greedy arrows. The start $v_0 = 0$ is optimistic, and *stay*, at −1 a step, looks cheap: at B, staying beats the −35 descent until the values have counted 34 stays.

Policy iteration needs two cycles, and the first improvement already gives the optimal policy π_A. Value iteration needs 36 sweeps, yet its greedy policy is optimal on the hiker's path, at S and A, after 5. The long tail is spent on B, a state the optimal hiker never visits: a case for the asynchronous updates of section 4.3.

All three algorithms of this page, on the gridworld and on harder grids, in one Space:

INTERACTIVE SPACE **Solving gridworlds by dynamic programming**

Policy evaluation, policy iteration and value iteration, one tab each, on the Sutton and Barto gridworld (Example 4.1) and on other presets: walls, a slippery cliff, a trap with no exit.

▶ Run it here Open in a new tab ↗

WHAT TO TRY in the Space above ›

In a few lines:

- Dynamic programming solves the Bellman equations when the model p is known, one sweep over the states at a time, with the value function as the cache of the subproblems.
- Policy evaluation iterates the expectation backup and converges to v_π; the order between the states settles long before the values.
- Policy improvement: acting greedily with respect to v_π never makes the policy worse, and changes nothing only when π is already optimal.
- Policy iteration alternates full evaluations and greedy improvements, and stops after a finite number of them: few iterations, each one expensive.
- Value iteration iterates the optimality backup: many sweeps, each one cheap. Modified policy iteration sits in between, and all three are generalized policy iteration.
- v_* is unique; the optimal policy need not be.

The next move, from exact and exhaustive (this page) to sampled, starts in Part 3.1: Learning from Experience and continues in Part 3.2: Learning to Act. There, the agent no longer knows p: Monte Carlo methods, temporal-difference learning, SARSA and Q-learning estimate the same values from the transitions they experience. The last step, from tables to approximate functions, is Part 4: From Discrete to Continuous, where these methods start solving real problems.

Extra exercises

Three exercises to go further, from the lightest to the most mathematical:

- E.1, the gambler's problem on three states: the optimality equation, checked by hand.
- E.2, the full gambler's problem: reasoning about an optimal policy, no new math.
- E.3, why a deterministic optimal policy always exists: a proof, built on the contraction of section 3.2.

Exercise E.1 The gambler's problem, reduced ●○○ Compute — Show ›

Exercise E.2 The gambler's curious optimal policy ●●○ Reason — Show ›

Exercise E.3 Why a deterministic optimal policy exists ●●● Prove — Show ›

Citation

For attribution in academic contexts, please cite this work as

```
Etienne Chassaing (2026). "Planning with a Model".
```

BibTeX citation

```
@misc{chassaing2026_planning_with_a_model,
  title={Planning with a Model},
  author={Etienne Chassaing},
  year={2026},

}
```

Reuse

Text and figures are licensed under CC-BY 4.0. Built with the Research Article Template (Thibaud Frère, CC-BY 4.0) and the exercise components of NLP before LLMs (David Louapre).

Made with ❤️ with research article template