Faculté des **sciences économiques** et de **gestion**
Université de Strasbourg

Reinforcement Learning

UE 2 Machine learning

Nattirat Mayer

Fall-Winter Semester 2025-26

We will cover...

Model-Based RL

Dynamic Programming
-Policy Iteration
-Value Iteration

Model-Free RL

Monte Carlo
-On-Policy MC
-Off-Policy MC

Model-Free RL

Temporal Difference
- TD(.) (on-policy)
-SARSA (on-policy)
-Q-Learning (off-policy)

Model-Free RL

Deep RL
- Deep Q-Learning (off-policy)
- Policy Gradient (on-policy)

*Not exclusive list

Chapter 4. Dynamic Programming

Here comes a new chapter!

4. Dynamic programming
   - 0. Revision of RL
   - 4.1 Value Iteration
   - 4.2 Policy Iteration
   - 4.3 Policy Iteration vs Value Iteration
   - 4.3 DP Tutorial
   - 4.4 Asynchronous dynamic programming
   - 4.5 Generalized policy iteration (GPI)
   - 4.6 Downsides of DP

0. Revision of Reinforcement Learning

Goal: To teach intelligent agents to learn how to perform tasks optimally.

To perform a task, the main challenge for the agent is to find the **optimal actions** to take.

The agent selects actions according to a policy $\pi(a \mid s)$, which is a distribution. When we run the policy through the MDP, we get a trajectory:

$$S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$$

We want the total discounted sum of rewards to be high. This discounted sum of rewards is denoted as G_t:

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

To achieve this, we consider the value functions:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$$

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

From

$$V(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$$

Recall from the previous chapter, the **Bellman Expectation Equation**:

$$V(s) = \mathbb{E}_\pi\left[R_{t+1} + \gamma V(S_{t+1}) \mid S_t = s\right]$$

For optimal decisions, we take the action a that maximizes the expected return. From the **Bellman Optimality Equation:**

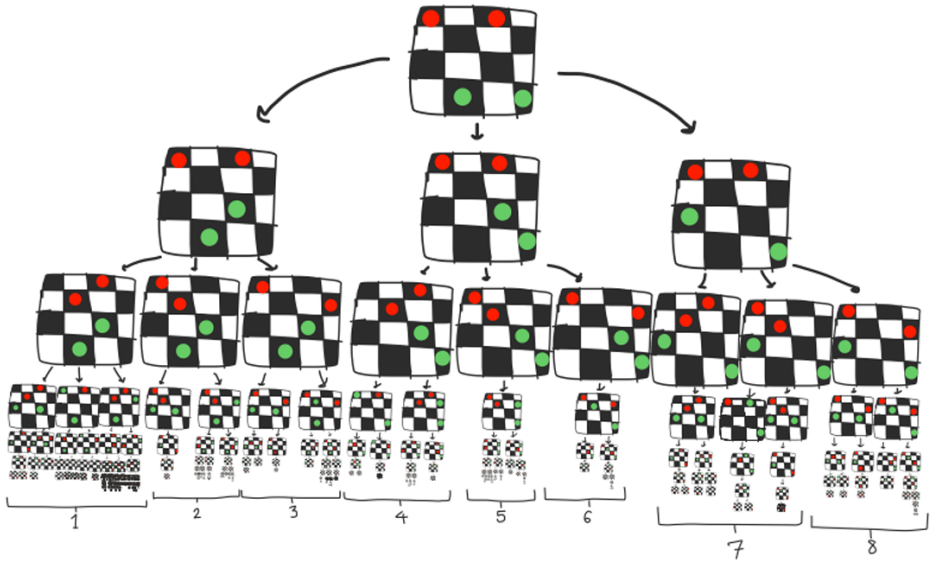
$$V^*(s) = \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V^*(s')\right]$$

These equations create a kind of *chicken-and-egg problem*. Which means to compute $V^*(s)$, we need to know $V^*(s')$, but $V^*(s')$ itself depends on other future states. In this chapter, we use **Dynamic Programming (DP)** to solve this.

4. Dynamic programming

What is Dynamic Programming (DP)?

Richard Bellman (1920-1984) *'How to solve multi-step optimization problem by breaking into smaller recursive* ***sub-problems****'*

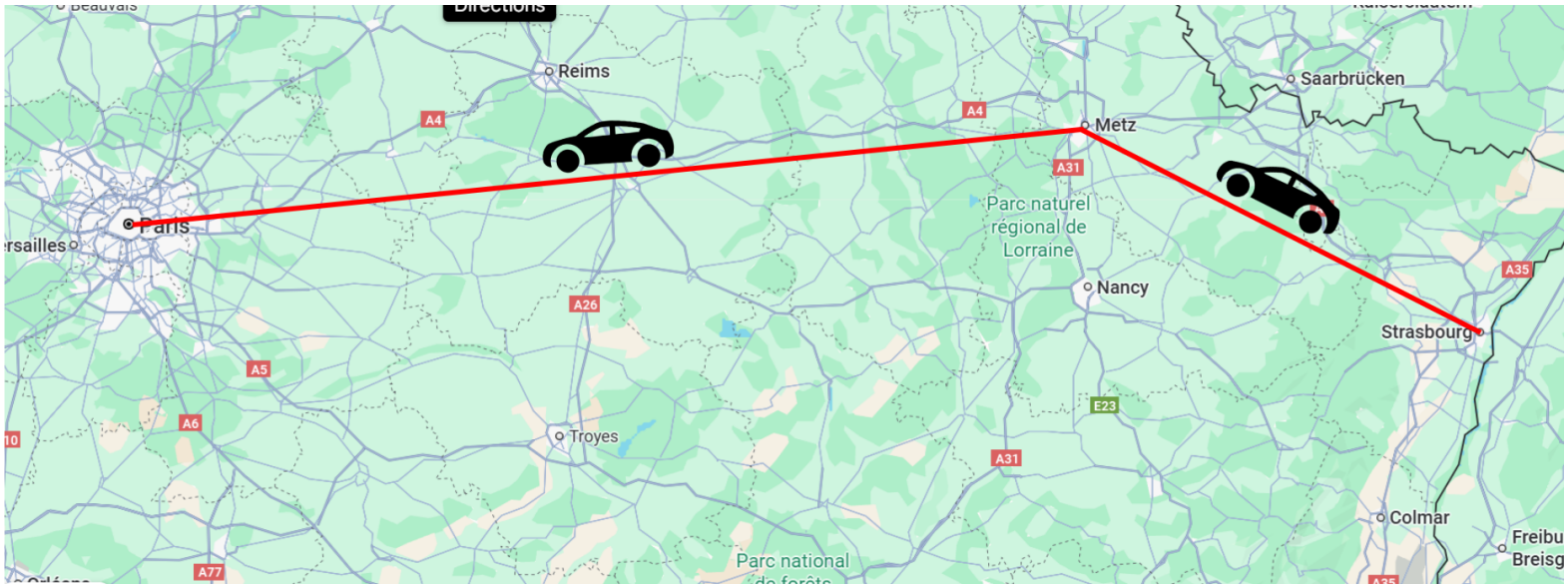


Reference: MIT Mathematics

- Solve subproblems
- Combine solutions to subproblems

Problem: From Strasbourg to Paris
Subproblems:

- Strasbourg $\rightarrow$ Metz
- Metz $\rightarrow$ Paris

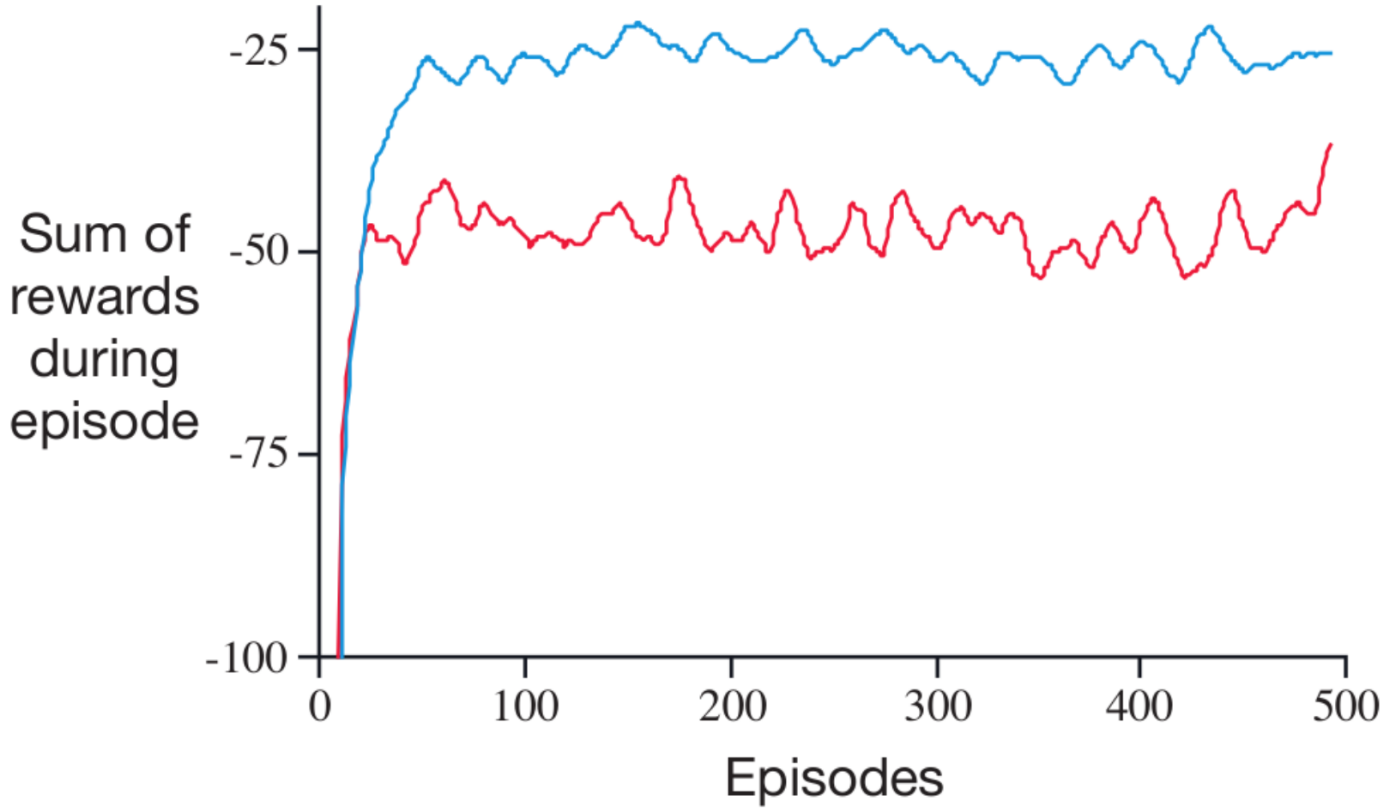


Dynamic Programming (DP) solves this by *iteratively* applying the Bellman equation until the value function $V(s)$ converges to $V^*(s)$.

Convergence means that repeated updates eventually stop changing the value function $V(s)$, i.e., the estimates become stable.
Example

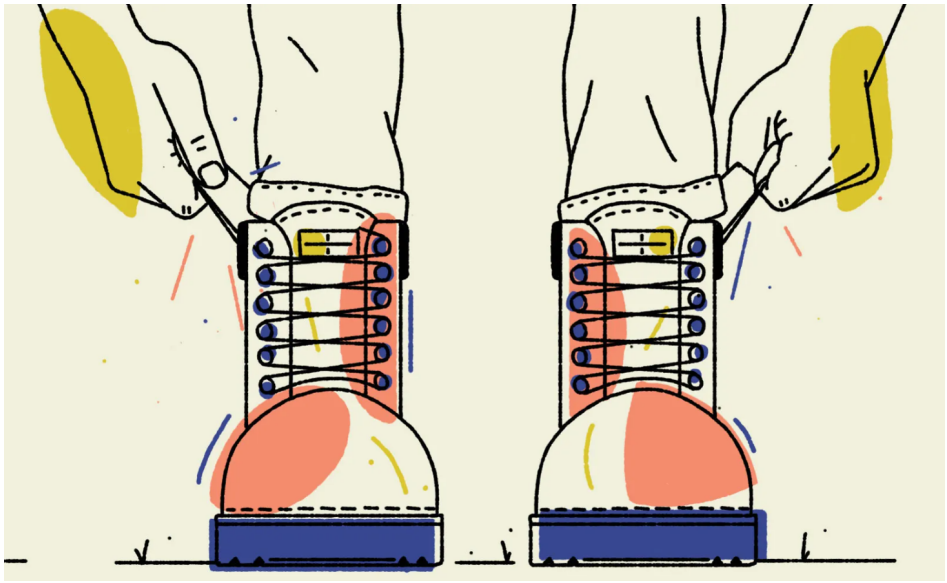
- Initially, $V(s)$ are just guesses
- After each Bellman update, values get closer to the true $V^*(s)$
- When changes become very small (below a threshold), we say it has **converged** (your agent has finished learning)

Convergence...



Rate of convergence is how fast your agent learn. In general, the faster = more efficient learning

Bootstrapping and DP

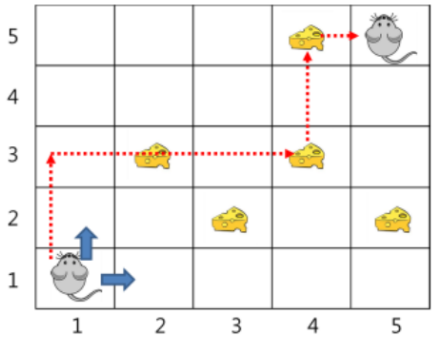


Reference: Huffpost, 2018

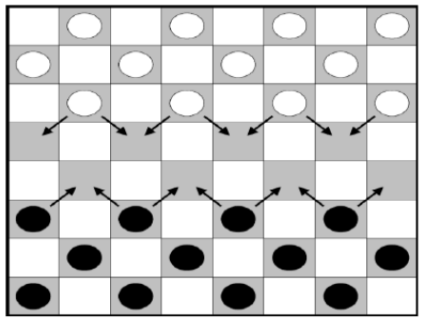
- Bootstrap = Making a guess from a previous guess
- Bootstrap = ”to pull yourself up by your own bootstraps”
- In RL, bootstrapping is to use your own current estimates to update other estimates
- Unlike Monte Carlo (next Chapter), DP uses **bootstrapping** to approximate the value functions

Dynamic Programming (DP) can also solve...

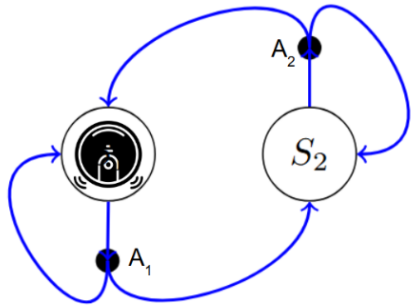
Mouse in a grid maze



Checker



Recycling robot



Tic Tac Toe

X	O	O
O	X	X
		X

Frozen Lake

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

- Scheduling algorithms
- String algorithms (e.g. sequence alignment)
- Graphical models (e.g. Viterbi algorithm-finds the most likely sequence of hidden events)
- Bioinformatics (e.g. lattice models)

Reference: Silver (2015)

4. Dynamic programming (DP)

Assumptions:

- The environment is according to **Markov Decision Processes (MDP)**, that is
  - We know the transition probability $p(s', r|s, a)$
  - We know the reward function $R(s, a)$

Two main algorithms of DP

- Value Iteration
- Policy Iteration
  - Policy Evaluation
  - Policy Improvement

4.1 Value Iteration

For Value Iteration, we update V(s) until they stop changing (convergence) with the Bellman Optimality

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

In python,

```
            for a in actions:
                v = 0
                for ns, p in step(s, a).items(): # ns is the next state s'
                    v += p * (reward(ns) + gamma * V[ns])
                vals.append(v)
            best_v = max(vals)
            delta = max(delta, abs(V[s] - best_v)) # delta to check if converged
            V[s] = best_v
            policy[s] = actions[np.argmax(vals)]
        delta_history.append(delta)
        if delta < theta: # check for convergence
            break
    return V, policy, delta_history
```

Value Iteration - Algorithm

1. Starts with a state value function $v(s) = 0$
2. Iteratively evaluates and improves on $v(s)$ by estimating the value of a state to be based on the action that will lead to maximal return
3. Repeats until convergence
4. Once converged, the optimal policy is trivially derived from $v(s)$

```
            for a in actions:
                v = 0
                for ns, p in step(s, a).items(): # ns is the next state s'
                    v += p * (reward(ns) + gamma * V[ns])
                vals.append(v)
            best_v = max(vals)
            delta = max(delta, abs(V[s] - best_v)) # delta to check if converged
            V[s] = best_v
            policy[s] = actions[np.argmax(vals)]
        delta_history.append(delta)
        if delta < theta: # check for convergence
            break
    return V, policy, delta_history
```

4.2 Policy Iteration

Policy Iteration alternates between two steps:

- **Policy Evaluation ($\mathbf{E}$):** Compute $v_\pi(s)$ for the current policy.
- **Policy Improvement ($\mathbf{I}$):** Update policy π to π' using $v_\pi(s)$.

Repeating these steps iteratively leads to the optimal policy and value function:

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

- For **Policy Evaluation**, we consider the Bellman value function for policy π which satisfies:

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

- For **Policy Improvement**, the agent acts greedily with respect to $v_\pi(s)$

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

By choosing actions this way, the new policy π' is guaranteed to be at least as good as the old policy π, i.e., $v_{\pi'}(s) \geq v_\pi(s)$ for all s.

For **Policy Evaluation**, the same equation as last slide,

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

```
# Policy Evaluation
while True:
    delta = 0
    for s in states:
        if s == goal or s in holes:
            continue
        a = policy[s]
        v = 0
        for ns, p in step(s, a).items():
            v += p * (reward(ns) + gamma * V[ns])
        delta = max(delta, abs(V[s] - v))
        V[s] = v
    if delta < theta:
        break
```

For **Policy Improvement**, the same equation as previously,

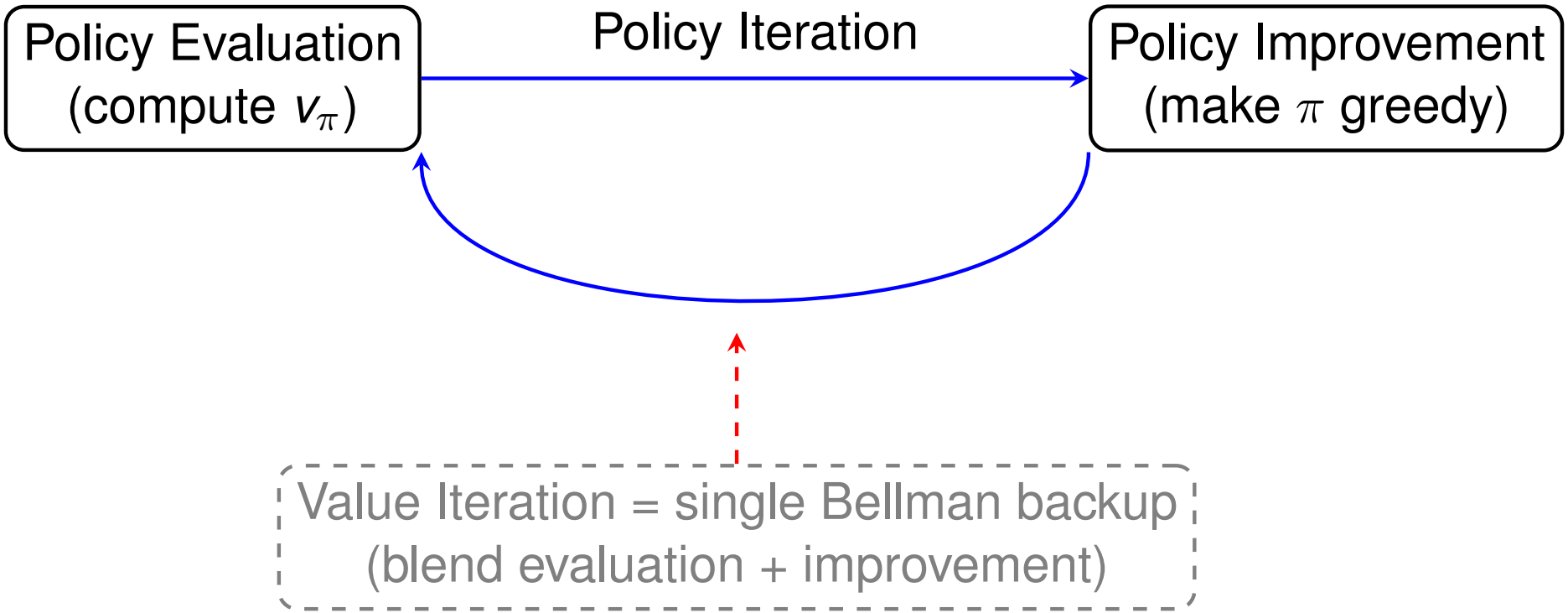
$$\pi'(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s') \right]$$

```
        # Policy Improvement
        stable = True
        for s in states:
            if s == goal or s in holes:
                continue
            old_a = policy[s]
            action_values = []
            for a in actions:
                q = 0
                for ns, p in step(s, a).items():
                    q += p * (reward(ns) + gamma * V[ns])
                action_values.append(q)
            best_a = actions[np.argmax(action_values)]
            policy[s] = best_a
            if best_a != old_a:
                stable = False
        stable_history.append(stable)
    return V, policy, stable_history
```

Policy Iteration - Algorithm

1. Starts with a state value function $v(s)$ and a given policy π
2. Iteratively evaluates $v(s)$ using the Bellman update for state value functions until $v(s)$ is an accurate representation of $v_\pi(s)$
3. Improves π by making it greedy with respect to $v(s)$
4. Repeats until convergence

4.3 Policy Iteration vs Value Iteration

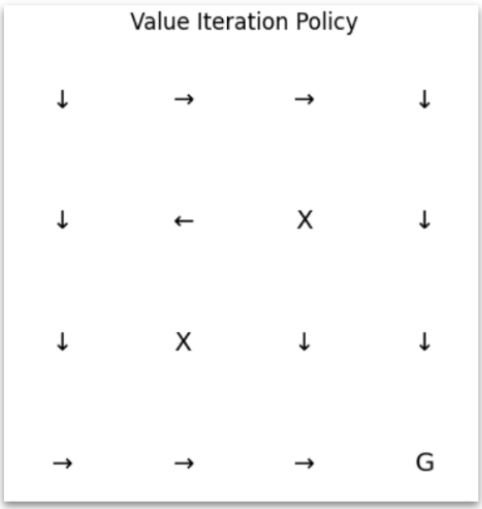


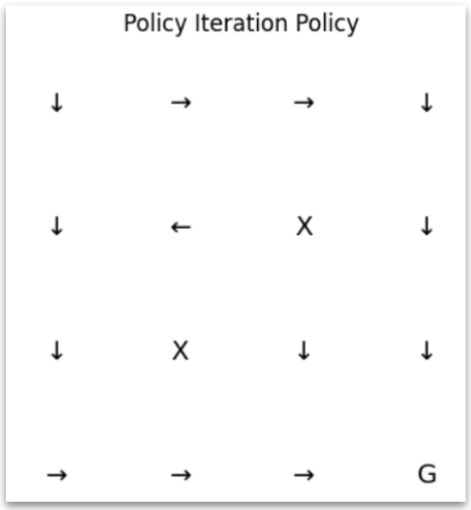
4.3 DP Tutorial

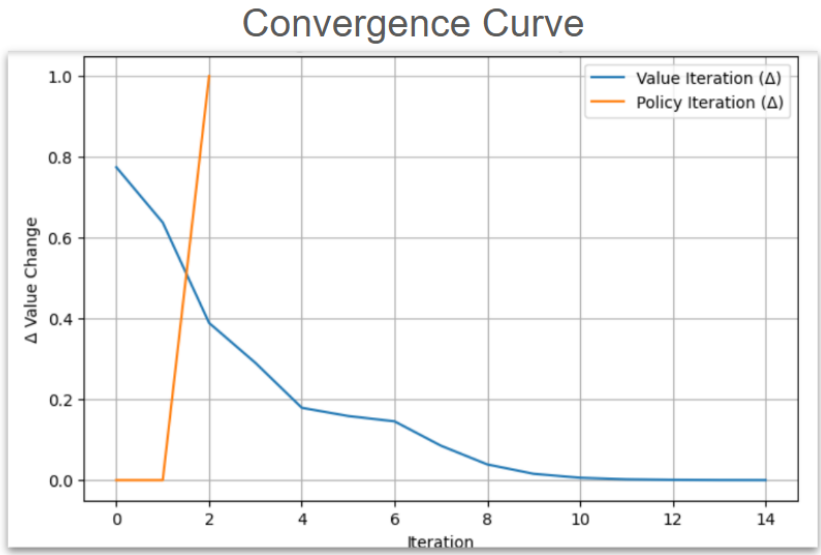
Frozen Lake Challenge: Value Iteration and Policy Iteration
File name: *dp-frozen-lake.ipynb*

- Consider the 4×4 gridworld
  - Discounted episodic MDP ($\gamma \in (0, 1)$)
  - The states: $\mathcal{S} = \{1, 2, \ldots, 16\}$.
  - The actions: $\mathcal{A} = \{\texttt{up}, \texttt{right}, \texttt{down}, \texttt{left}\}$
  - Reward $R_t = +1$ for goal, $R_t = -1$ for in holes, $R_t = -0.04$ for any other states
  - Currently, the grid has two holes as shown below

Run file *dp-frozen-lake.ipynb*
Based-line result







Exercises: Fine-tuning Policy & Value Iteration (Part 1)

Try modifying the Frozen Lake code to explore how the agent's behavior changes:

1. Change the discount factor γ (e.g., 0.5, 0.9, 0.99) and observe its effect on the value function and policy.
2. Adjust the slip probability (e.g., 0.0, 0.1, 0.3) and see how stochasticity affects convergence.
3. Modify the step penalty (currently $= -0.04$) to see how it influences shortest paths to the goal.

Exercises: Fine-tuning Policy & Value Iteration (Part 2)

Continue exploring the Frozen Lake code:

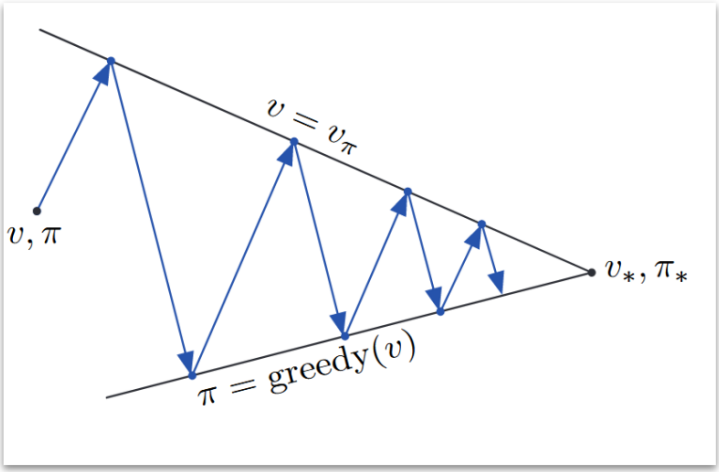
4. Combine changes: for example, high γ and high slip probability. What patterns emerge in the policy?
5. Compare the number of iterations until convergence for value iteration vs. policy iteration under different parameter settings.

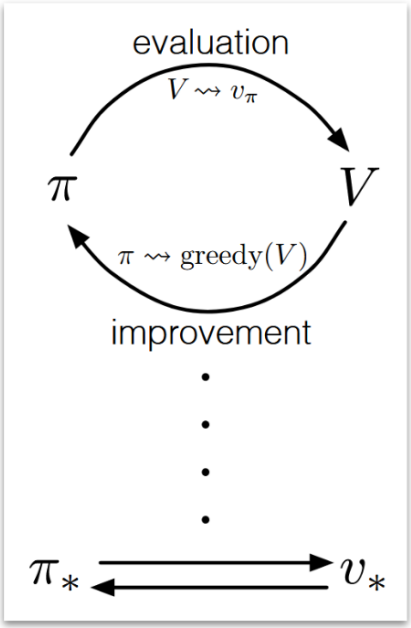
Value Iteration Vs Policy Iteration

4.4 Asynchronous dynamic programming

- One drawback of former DP methods is that the state set can be very large and iterative computation of the value and policy can become time consuming.
- Asynchronous DP algorithms are not systematically updating the values of all states of the state set, but only those few of them, those which are the most relevant.

4.5 Generalized policy iteration (GPI)





- **Policy evaluation**
  Estimate v_π
  (using any policy-evaluation algorithm)
- **Policy improvement**
  Generate $\pi' \leftarrow \text{improve}(\pi)$ (e.g., greedy wrt. v_π)
  (using any policy-improvement algorithm)

4.6 Downsides of DP

- Need to know transition probability $p(s', r|s, a)$ and reward function $r(s, a)$
- The *curse of dimensionality*, when state space $\mathcal{S}$ becomes large. DP breaks down.