

Problem set 2 - Monte Carlo method & TD learning

Due: 11:59pm, October 10, 2025

Problem 1 - Monte Carlo evaluation and Monte Carlo control (Coding).

TODO list:

- (1) Finish MC Policy Evaluation Code (10 pt)
- (2) Try different step size and discuss (10 pt)
- (3) Finish MC + Exploring Starts Code (10 pt)
- (4) Plot the convergence behavior, discuss the plot (10 pt)
- (5) Proof of Monte-Carlo control on *random walk* problem (20 pt)
- (6) Finish MC + epsilon greedy Code (10 pt)

Problem 2 - Temporal-difference evaluation, SARSA and Q-learning

- (1) Finish TD evaluation Code (10 pt)
- (2) Finish SARSA algorithm (10 pt)
- (3) Finish Q-learning algorithm (10 pt)

Problem 1: Monte-Carlo method on CliffWalk environment.

Recall from last PSET, the **CliffWalking** gridworld is a 4×12 grid. The agent starts at the bottom-left cell and aims to reach the bottom-right. The bottom row between start and goal is a **cliff**; stepping into it ends the episode with a large penalty. Each non-terminal step yields -1 ; stepping into the cliff yields -100 (and termination).

In this problem we will **directly use Gym/Gymnasium** to interact with the environment (`CliffWalking-v1`) and perform **Monte-Carlo (MC) method**. Here we provide the print utility function as the same in PSET 1.

```
In [1]: import numpy as np
import random
import gymnasium as gym
from tqdm import tqdm
import matplotlib.pyplot as plt

# ----- Reproducibility -----
seed = 0
np.random.seed(seed)           # numpy RNG (env seeding can be done at reset time)

# ----- Environment -----
env = gym.make("CliffWalking-v1") # 4x12 grid; bottom row is cliff + goal

# Pretty printing for small numeric tables
```

```

np.set_printoptions(precision=3, suppress=True)

def print_values(values, nrow: int, ncol: int, title: str = "State Values"):
    """
    Print a value table V in grid form.
    """
    values = np.asarray(values).reshape(nrow, ncol)
    print(title)
    for r in range(nrow):
        print(" ".join(f"{values[r, c]:.2f}" for c in range(ncol)))
    print()

def print_policy(pi, nrow: int, ncol: int, title: str = "Policy"):
    """
    Render a policy on the CliffWalking grid as arrows.

    The environment uses the action indexing:
    0 = UP, 1 = RIGHT, 2 = DOWN, 3 = LEFT
    """
    arrow = {0: "^", 1: ">", 2: "v", 3: "<"} # matches env action semantics
    print(title)
    for i in range(nrow):
        row_syms = []
        for j in range(ncol):
            s = i * ncol + j
            p = pi[s]

            # Determine greedy action(s)
            if isinstance(p, list) and len(p) == 4:
                best = np.argwhere(np.array(p) == np.max(p)).flatten().tolist()
            elif isinstance(p, int):
                best = [p]
            else:
                arr = np.array(p, dtype=float).ravel()
                best = np.argwhere(arr == np.max(arr)).flatten().tolist()

            # Cliff/goal cells (bottom row except column 0) rendered as terminal
            if i == nrow - 1 and j > 0:
                row_syms.append("T")
            else:
                row_syms.append("".join(arrow[a] for a in best))
        print(" ".join(sym if sym else "." for sym in row_syms))
    print()

# ----- Reference table (ground-truth under random policy from PSET1) -----
# Shape is (4, 12) in row-major order, then flattened to 1D for convenience.
# Values correspond to the state-value function  $V^\pi$  for the UNIFORM RANDOM policy.
V_random_gt = np.array([
    [-143.21, -147.36, -151.35, -153.93, -155.11, -155.05, -153.66, -150.44, -144.43, -134.39, -124.35, -114.31],
    [-164.99, -174.34, -180.41, -183.52, -184.80, -184.82, -183.62, -180.68, -174.67, -162.95, -152.93, -142.91],
    [-207.96, -237.09, -246.20, -249.36, -250.43, -250.52, -249.79, -247.81, -243.17, -231.62, -221.59, -211.57],
    [-261.35, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00]
], dtype=float).flatten()

```

1.1 Monte-Carlo evaluation

Let $\mathcal{D}(s)$ denote the set of all time indices at which state s is visited across sampled episodes. Then the Monte Carlo estimate of the value function is

$$\hat{V}(s) = \frac{1}{|\mathcal{D}(s)|} \sum_{t \in \mathcal{D}(s)} g_t. \quad (2.3)$$

There are two common variants:

- **First-visit MC:** use only the first occurrence of s in each episode.
- **Every-visit MC:** use all occurrences of s within an episode.

You can test both of them in 1.2. For 1.1 you need to finish the **first-visit** one.

TODO: Finish the code block for monte-carlo evaluation.

```
In [2]: def mc_evaluate(policy, env, episodes=5000, gamma=0.95, seed=None):
    """
    First-visit Monte Carlo (MC) state-value prediction (Sutton & Barto, Alg. 5.1).

    Parameters
    -----
    policy : ndarray, shape (nS, nA)
        Row-stochastic policy: for each state s, policy[s] is a prob. dist. over actions.
    env : Gymnasium-like environment
        Must expose discrete observation_space.n and action_space.n and return
        (obs, reward, terminated, truncated, info) from step().
    episodes : int
        Number of episodes to sample.
    gamma : float
        Discount factor in [0, 1].
    seed : int | None
        If given, used to seed a NumPy RNG and (re)seed env at each episode start.
    """
    nS, nA = env.observation_space.n, env.action_space.n
    V = np.zeros(nS, dtype=float)          # value estimates
    N_first = np.zeros(nS, dtype=int)       # first-visit counts per state
    visits = np.zeros(nS, dtype=int)        # total visits (diagnostic only)
    errors = []

    rng = np.random.default_rng(seed)

    for ep in tqdm(range(episodes), desc="MC first-visit"):
        # Episode generation under π
        s, _ = env.reset(seed=int(rng.integers(1e9)) if seed is not None else None)
        states, rewards = [], []
        done = False

        while not done:
            ##### # Record trajectory
            # TODO: sample action from policy and step in env
            # hint: use env.step(a) to get (s', r, terminated, truncated, info)

            a = rng.choice(nA, p=policy[s])  # sample action from policy
            s_next, r, terminated, _ = env.step(a)

            # Record trajectory
            states.append(s)
            rewards.append(r)
            visits[s] += 1
```

```

s = s_next
done = terminated or truncated
#####
# Identify first visits of each state in this episode
first_visit_mask = [False] * len(states)
seen_from_start = set()
for t, st in enumerate(states):
    if st not in seen_from_start:
        seen_from_start.add(st)
        first_visit_mask[t] = True

# Backward return accumulation; update ONLY on first visits
#####
# TODO: update V and N_first for first-visit states
G = 0.0
for t in reversed(range(len(states))):
    G = gamma * G + rewards[t]
    if first_visit_mask[t]:
        st = states[t]
        N_first[st] += 1
        # incremental mean update
        V[st] += (G - V[st]) / N_first[st]
#####

# Track RMSE to a reference table for the random policy (from PSET1)
rmse = float(np.sqrt(np.mean((V - V_random_gt) ** 2)))
errors.append(rmse)

return V, errors

# Uniform-random policy for evaluation
policy = np.ones((env.observation_space.n, env.action_space.n)) / env.action_space.n

V, errors = mc_evaluate(policy, env, episodes=1000)

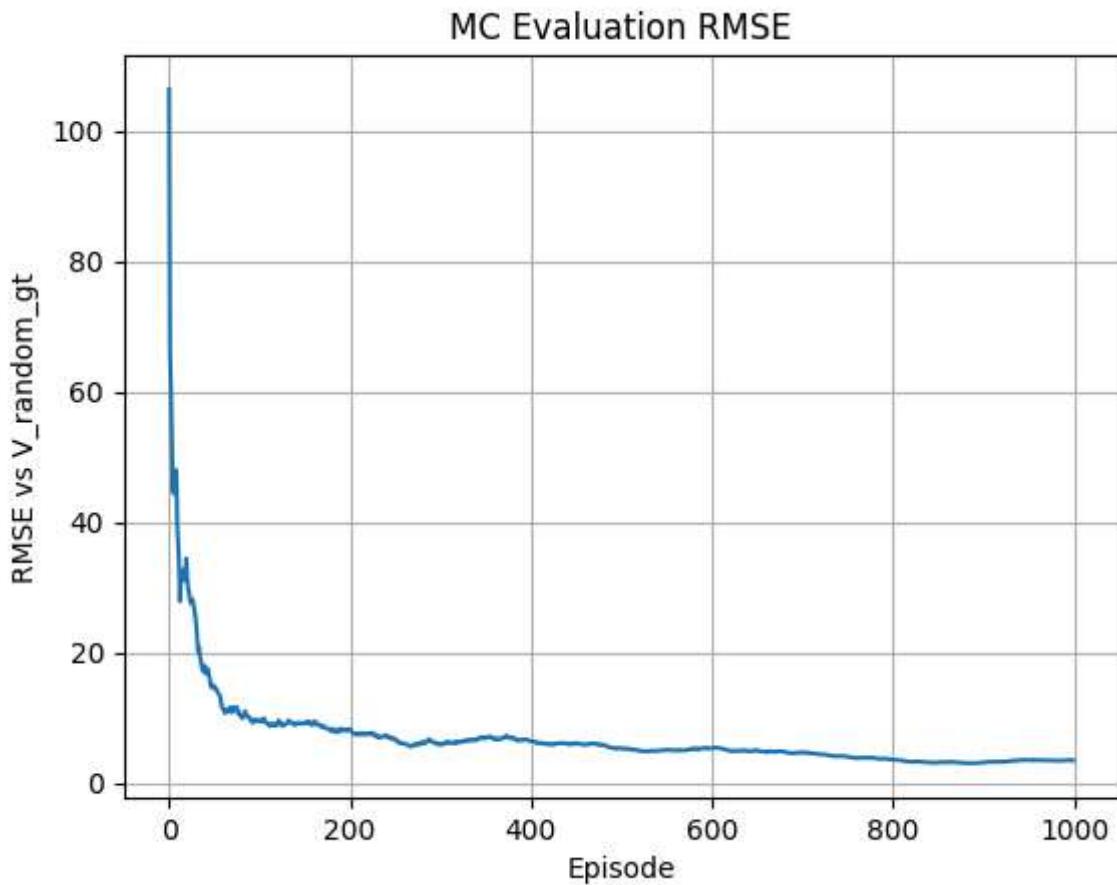
print_values(V, 4, 12, "Estimated State Values (MC First-Visit)")
# Compare visually with PSET1 reference; they should be close (up to sampling noise).

plt.figure()
plt.plot(errors)
plt.xlabel("Episode")
plt.ylabel("RMSE vs V_random_gt")
plt.title("MC Evaluation RMSE")
plt.grid(True)
plt.show()

# Note: On my laptop, this may take ~1 minute for 1k episodes, depending on Python + Gym versions

```

```
MC first-visit: 100%|██████████| 1000/1000 [04:29<00:00, 3.70it/s]
Estimated State Values (MC First-Visit)
-147.83 -151.14 -150.53 -157.76 -155.98 -148.27 -148.53 -145.86 -137.13 -127.30 -123.37 -103.59
-169.74 -173.24 -182.10 -185.20 -187.96 -182.43 -183.04 -178.44 -167.13 -159.39 -144.20 -111.66
-209.25 -241.05 -252.15 -255.00 -253.18 -247.93 -247.16 -252.20 -238.69 -230.99 -200.20 -100.53
-261.65  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
```



1.2 Plot for different step size.

Recall equation (2.4) in the lecture note:

$$\hat{V}(s) \leftarrow \hat{V}(s) + \alpha_{N(s)} (g_t - \hat{V}(s)), \quad \alpha_{N(s)} > 0 \text{ diminishing.}$$

For standard Monte-Carlo evaluation, you will take the $\alpha_{N(s)}$ as $\frac{1}{N(s)}$. But you have more options: you can do constant step or other steps you prefer.

TODO: test different step size options (sample average, constant step size with different values, and another step size schedule that satisfies Robbins-Monro), and you can also change first-visit to every-visit. What do you observe?

Hint: you can try save different loss curves and plot in the same plot

```
In [4]: from collections import defaultdict
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt

def mc_evaluate_flexible(policy, env, episodes=5000, gamma=0.95, seed=None,
                        step_rule="sample-average", step_params=None,
                        visit_type="first"):
    """
    Generalized MC state-value prediction supporting different step-sizes and visit types.

    step_rule:
        - "sample-average": alpha_n(s) = 1 / N_updates_for_s
    """

    # Initialize state-value function
    V = defaultdict(lambda: 0.0)
    N = defaultdict(lambda: 0.0)

    # Initialize step-size parameters
    if step_params is None:
        step_params = {
            "sample-average": 1.0 / episodes
        }
    else:
        step_params["sample-average"] = step_params.get("sample-average", 1.0 / episodes)

    # Initialize visit type
    if visit_type == "first":
        visit_fn = first_visit
    elif visit_type == "every":
        visit_fn = every_visit
    else:
        raise ValueError(f"Unknown visit type: {visit_type}")

    # Initialize progress bar
    pbar = tqdm(total=episodes)

    for episode in range(episodes):
        # Reset environment
        state = env.reset()

        # Initialize visit history
        visit_history = []

        while True:
            # Select action based on policy
            action = policy.select_action(state)

            # Take action and get reward
            next_state, reward, done, _ = env.step(action)

            # Record visit
            visit_fn(state, action, next_state, reward, V, N, step_params)

            # Move to next state
            state = next_state

            # Check if episode is done
            if done:
                break

        # Update progress bar
        pbar.update(1)

    # Plot results
    plt.figure(figsize=(10, 6))
    plt.plot(V)
    plt.title("State-Value Function (V-hat(s))")
    plt.xlabel("State (s)")
    plt.ylabel("Value Function (V-hat(s))")
    plt.show()

```

```

        - "constant":  $\alpha$  (fixed) -> set step_params={"alpha": 0.1}
        - "power":  $\alpha_n(s) = c / (n + n_0)^p$  (Robbins-Monro if  $0.5 < p \leq 1$ )
                    set step_params={"c":1.0, "p":0.7, "n0":0}

visit_type:
    - "first": first-visit MC updates
    - "every": every-visit MC updates
"""
step_params = step_params or {}
nS, nA = env.observation_space.n, env.action_space.n
V = np.zeros(nS, dtype=float)
N_updates = np.zeros(nS, dtype=int) # counts how many times we've UPDATED a state's V
errors = []

rng = np.random.default_rng(seed)

def alpha_for_state(s):
    if step_rule == "sample-average":
        # Use 1 / N_updates(s) *after* incrementing N_updates.
        return 1.0 / N_updates[s]
    elif step_rule == "constant":
        return float(step_params.get("alpha", 0.1))
    elif step_rule == "power":
        c = float(step_params.get("c", 1.0))
        p = float(step_params.get("p", 0.7)) # Robbins-Monro if  $0.5 < p \leq 1$ 
        n0 = float(step_params.get("n0", 0.0))
        # Use current count (after increment) for  $\alpha_n$ 
        n = N_updates[s]
        return c / ((n0 + n) ** p)
    else:
        raise ValueError(f"Unknown step_rule: {step_rule}")

for _ in tqdm(range(episodes), desc=f"MC ({visit_type}, {step_rule})"):
    s, _ = env.reset(seed=int(rng.integers(1e9)) if seed is not None else None)
    states, rewards = [], []
    done = False

    # Generate an episode
    while not done:
        a = rng.choice(nA, p=policy[s])
        s_next, r, terminated, truncated, _ = env.step(a)
        states.append(s)
        rewards.append(r)
        s = s_next
        done = terminated or truncated

    # First-visit mask
    if visit_type == "first":
        first_visit_mask = [False] * len(states)
        seen = set()
        for t, st in enumerate(states):
            if st not in seen:
                seen.add(st)
                first_visit_mask[t] = True
    else:
        # every-visit: update at ALL time steps
        first_visit_mask = [True] * len(states)

    # Backward return accumulation; update at selected visits

```

```

G = 0.0
for t in reversed(range(len(states))):
    G = gamma * G + rewards[t]
    if first_visit_mask[t]:
        st = states[t]
        # increment update counter BEFORE computing alpha so alpha uses n>=1
        N_updates[st] += 1
        alpha = alpha_for_state(st)
        V[st] += alpha * (G - V[st])

rmse = float(np.sqrt(np.mean((V - V_random_gt) ** 2)))
errors.append(rmse)

return V, np.array(errors)

# ----- Experiments: step sizes x visit types -----
# Uniform-random policy for evaluation (as in your snippet)
policy = np.ones((env.observation_space.n, env.action_space.n)) / env.action_space.n

configs = [
    # First-visit
    ("First/SampleAvg", dict(step_rule="sample-average", visit_type="first")),
    ("First/Const 0.1", dict(step_rule="constant", step_params={"alpha":0.1}, visit_type="first")),
    ("First/Const 0.02", dict(step_rule="constant", step_params={"alpha":0.02}, visit_type="first")),
    ("First/RM p=0.7", dict(step_rule="power", step_params={"c":1.0, "p":0.7, "n0":0}, visit_type="first")),
    # Every-visit
    ("Every/SampleAvg", dict(step_rule="sample-average", visit_type="every")),
    ("Every/Const 0.1", dict(step_rule="constant", step_params={"alpha":0.1}, visit_type="every")),
    ("Every/RM p=0.7", dict(step_rule="power", step_params={"c":1.0, "p":0.7, "n0":0}, visit_type="every"))
]

results = {}
episodes = 1000
gamma = 0.95
seed = 42

for label, kwargs in configs:
    _, errs = mc_evaluate_flexible(policy, env,
                                    episodes=episodes,
                                    gamma=gamma,
                                    seed=seed,
                                    **kwargs)
    results[label] = errs

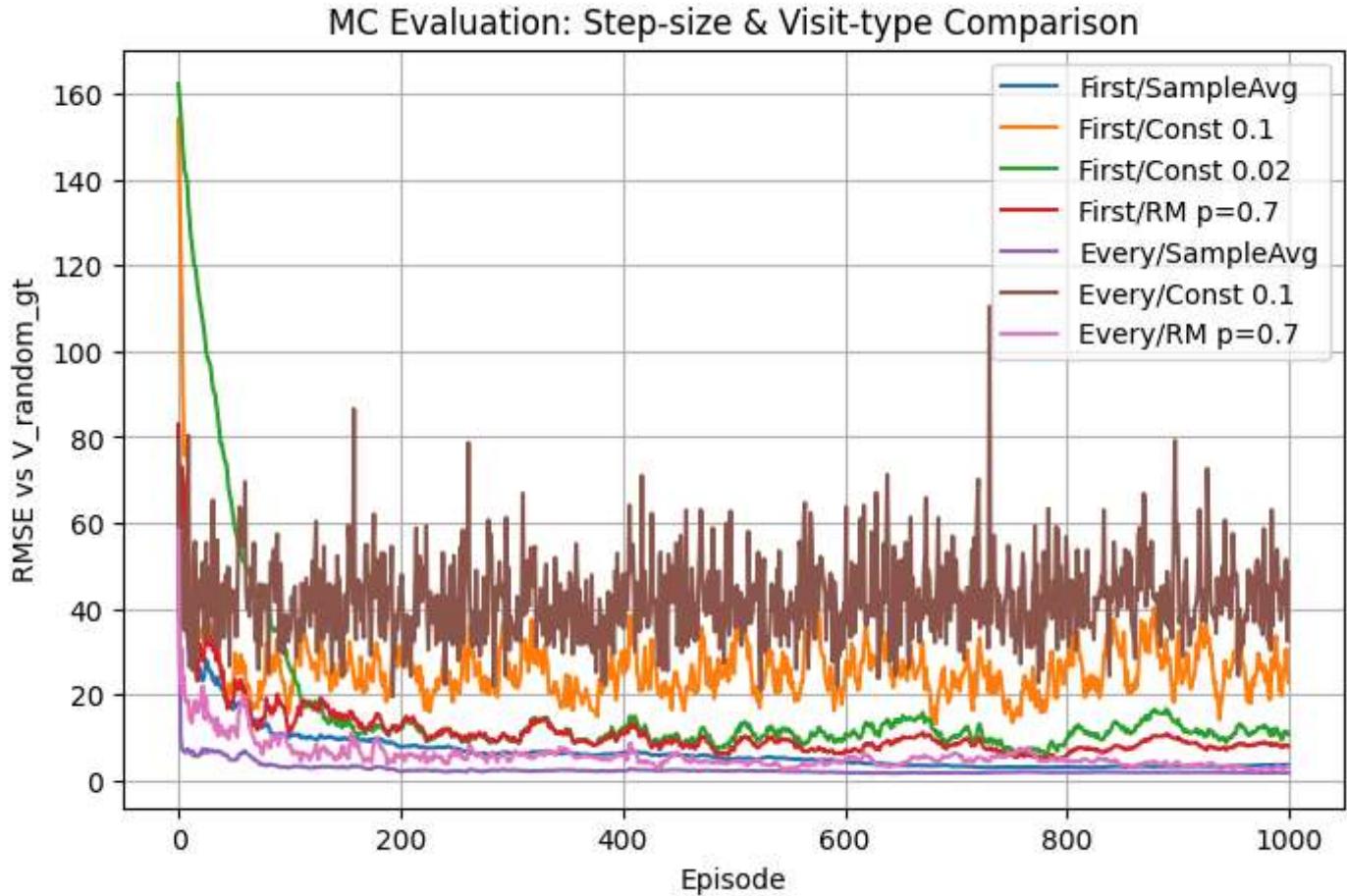
# ----- Plot -----
plt.figure(figsize=(8,5))
for label, errs in results.items():
    plt.plot(errs, label=label)
plt.xlabel("Episode")
plt.ylabel("RMSE vs V_random_gt")
plt.title("MC Evaluation: Step-size & Visit-type Comparison")
plt.grid(True)
plt.legend()
plt.show()

```

```

MC (first, sample-average): 100%|██████████| 1000/1000 [04:29<00:00, 3.72it/s]
MC (first, constant): 100%|██████████| 1000/1000 [04:33<00:00, 3.65it/s]
MC (first, constant): 100%|██████████| 1000/1000 [04:30<00:00, 3.69it/s]
MC (first, power): 100%|██████████| 1000/1000 [04:20<00:00, 3.84it/s]
MC (every, sample-average): 100%|██████████| 1000/1000 [04:47<00:00, 3.48it/s]
MC (every, constant): 100%|██████████| 1000/1000 [04:09<00:00, 4.01it/s]
MC (every, power): 100%|██████████| 1000/1000 [03:58<00:00, 4.19it/s]

```



We tested multiple step-size schedules for Monte Carlo policy evaluation, including sample-average ($\alpha_{N(s)} = 1/N(s)$), constant step sizes, and a Robbins-Monro schedule ($\alpha_n = \frac{c}{(n+n_0)^p}$, $0.5 < p \leq 1$). We also compared first-visit versus every-visit updates. The results can be summarized as follows:

Sample-average (diminishing step size):

This rule produces unbiased estimates and converges stably to the true value function. However, the learning is slow in the early episodes since the step size decreases as the number of visits increases. With more episodes, the RMSE decreases steadily toward zero.

Constant step sizes (e.g. $\alpha = 0.1$, $\alpha = 0.02$):

Larger constant step sizes yield faster initial reduction in error, but the estimates fluctuate around the true value function and do not converge exactly. A smaller constant step size leads to slower learning but more stable long-term behavior. This creates a trade-off between speed of learning and steady-state variance.

Robbins-Monro schedule:

Using a power-law decay (with $0.5 < p \leq 1$) combines the benefits of both approaches. It allows larger step sizes in the early stages for rapid error reduction, while still decaying sufficiently to guarantee convergence.

In practice, this schedule reduced RMSE faster than pure sample-average and achieved lower error than constant step sizes.

First-visit vs. Every-visit MC:

Every-visit updates generally converge faster than first-visit updates because more samples are used per episode. The variance is reduced by exploiting all occurrences of a state, while first-visit only uses one update per state per episode.

Sample-average MC is theoretically sound and convergent, but may be slow. Constant step sizes learn quickly but oscillate around the true values. Robbins-Monro schedules strike a good balance between bias and variance, making them practically attractive. Every-visit MC typically accelerates learning compared to first-visit MC.

1.3 Monte-Carlo control with Exploring Starts

In the class we talk about two methods to ensure exploration: exploring starts and ϵ -greedy policy. We will start with exploring starts for monte-carlo control.

TODO: finish code of Monte-Carlo control with exploring starts.

In [5]:

```
def _set_state(env, s):
    """Attempt to set the environment's internal discrete state to `s`
    (needed for Exploring Starts).
    """
    try:
        # Many classic-control/grid envs expose `unwrapped.s`
        env.unwrapped.s = s
        return True
    except Exception:
        # Some variants use `.state` instead
        try:
            env.unwrapped.state = s
            return True
        except Exception:
            return False

def mc_control_exploring_starts(env, episodes=5000, gamma=0.95, max_steps=500, alpha=None):
    """
    Monte Carlo control with Exploring Starts (ES). See Sutton & Barto, Alg. 5.3.

    Procedure
    -----
    1) For each episode, sample an initial state-action pair ( $s_0, a_0$ ) at random so that
       every  $(s, a)$  has nonzero probability.
    2) Roll out the *entire* episode under the current policy  $\pi$  (use  $a_0$  at  $t=0$ ; thereafter
       sample actions from  $\pi$ ).
    3) At the end of the episode, perform **first-visit** MC updates to  $Q(s, a)$  using the
       return  $G$  from each first-visited  $(s, a)$ .
    4) For all states visited in this episode, improve the policy greedily:
        $\pi(s) \leftarrow \text{argmax}_a Q(s, a)$  (break ties uniformly).
    """
    nS, nA = env.observation_space.n, env.action_space.n
    Q = np.zeros((nS, nA), dtype=float)
```

```

N_visits = np.zeros((nS, nA), dtype=int)
V_snaps = []

# Initialize  $\pi$  arbitrarily: start with a uniform distribution over actions.
policy = np.ones((nS, nA), dtype=float) / nA

for ep in tqdm(range(episodes), desc="MC ES (Exploring Starts)"):
    # --- Exploring Start: choose  $(s_0, a_0)$  uniformly at random ---
    # For CliffWalking we avoid terminal/cliff cells by excluding the last row
    # except the start (index 36). `nS - 11` ensures  $s_0 \in \{0..36\}$ .
    s0 = np.random.randint(nS - 11)
    a0 = np.random.randint(nA)

    # Reset env, then force the start state to  $s_0$ . If not supported, ES cannot be run strictly.
    s, _ = env.reset()
    if not _set_state(env, s0):
        raise RuntimeError(
            "Environment does not support setting arbitrary start states; "
            "strict Exploring Starts is not possible."
        )

    # --- Generate a full trajectory under current  $\pi$  ( $a_0$  at  $t=0$ , then follow  $\pi$ ) ---
    states, actions, rewards = [], [], []
    s = s0
    a = a0
    done = False

    # Cap Length to avoid pathological loops when  $\pi$  does not reach a terminal.
    for _ in range(max_steps):
        ######
        # TODO: sample action from policy and step in env
        # hint: use env.step(a) to get (s', r, terminated, truncated, info)
        # record current state-action
        states.append(s)
        actions.append(a)

        # step environment
        s_next, r, terminated, truncated, _ = env.step(a)
        rewards.append(r)
        done = terminated or truncated

        if done:
            break

        # follow policy thereafter
        a = np.random.choice(nA, p=policy[s_next])
        s = s_next
        #####

```

--- First-visit masks for (s, a) within this episode ---

```

first_visit_sa_mask = [False] * len(states)
seen_from_start_sa = set()
for t, (st, at) in enumerate(zip(states, actions)):
    if (st, at) not in seen_from_start_sa:
        seen_from_start_sa.add((st, at))
        first_visit_sa_mask[t] = True

```

--- Backward return accumulation; update Q only at first visits ---

```

#####

```

```

# TODO: update Q and N_first for first-visit states
G = 0.0
for t in reversed(range(len(states))):
    G = gamma * G + rewards[t]
    if first_visit_sa_mask[t]:
        st, at = states[t], actions[t]
        N_visits[st, at] += 1

    if alpha is None:
        # incremental mean
        Q[st, at] += (G - Q[st, at]) / N_visits[st, at]
    else:
        # constant step-size
        Q[st, at] += alpha * (G - Q[st, at])
#####
# --- Greedy policy improvement on states encountered this episode (uniform tie-breaking)
#####
# TODO: update policy to be greedy wrt. Q for states visited in this episode
for st in set(states):
    # find greedy action(s)
    greedy_actions = np.argwhere(Q[st] == Q[st].max()).flatten()
    # break ties uniformly
    new_pi = np.zeros(nA)
    new_pi[greedy_actions] = 1.0 / len(greedy_actions)
    policy[st] = new_pi
#####

# Keep a few early snapshots of V(s) for quick diagnostics
if (ep % 100) == 0 and ep <= 10000:
    V = Q.max(axis=1)
    V_snaps.append(V.copy())

return policy, Q, V_snaps

```

policy_es, Q_es, V_snaps = mc_control_exploring_starts(env, episodes=200000, gamma=0.95)

V_es = (policy_es * Q_es).sum(axis=1)

print_values(V_es, 4, 12, "V from MC-ES policy")
print_policy(policy_es, 4, 12, "MC-ES derived policy")

Note: With 200k episodes this can take around a minute on a typical Laptop
(depends on Python/Gym versions and hardware).

MC ES (Exploring Starts): 100% |██████████| 200000/200000 [01:57<00:00, 1695.16it/s]

V from MC-ES policy

-10.39	-9.80	-9.28	-8.86	-8.04	-7.41	-6.74	-6.04	-5.30	-4.53	-3.71	-2.85
-9.89	-9.38	-8.83	-8.07	-7.40	-6.73	-6.03	-5.30	-4.53	-3.71	-2.85	-1.95
-9.33	-8.71	-8.13	-7.50	-6.85	-6.04	-5.30	-4.52	-3.71	-2.85	-1.95	-1.00
-10.49	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

MC-ES derived policy

```

v > > > v > > v v > > v
> > v > > > v > > v v
> > > > > > > > > v
^ T T T T T T T T T T T

```

1.4 Convergence behavior

Use the `V_snap` you get from Monte-Carlo control to plot the convergence evolution.

TODO: what do you observe form the plot?

Hint: which part converge first?

```
In [6]: import matplotlib.animation as animation
from IPython.display import Image, display

# Basic check
V_snaps = [np.asarray(v).ravel() for v in V_snaps]
K = int(V_snaps[0].size)
if K != 48:
    raise ValueError(f"Expected V.size == 48 to reshape into 4x12, but got {K}.")

# Fixed color scale to avoid per-frame jumps (do not specify a colormap; use defaults)
V_all = np.vstack(V_snaps)
vmin, vmax = -20, 0

# Initialize figure: 2D grid
fig, ax = plt.subplots()
grid0 = V_snaps[0].reshape(4, 12)
im = ax.imshow(grid0, vmin=vmin, vmax=vmax) # Do not specify cmap
ax.set_xlabel("col")
ax.set_ylabel("row")

def init():
    im.set_data(grid0)
    return (im,) # blit requires a tuple

def update(i):
    grid = V_snaps[i].reshape(4, 12)
    im.set_data(grid)
    return (im,)

ani = animation.FuncAnimation(
    fig, update, frames=len(V_snaps), init_func=init, blit=True, interval=400, repeat=False
)

gif_path = "./mc_es_grid_evolution.gif"
ani.save(gif_path, writer="pillow")
plt.close(fig)

display(Image(filename=gif_path))
print("Saved GIF to:", gif_path)

# Print the last frame as a 4x12 grid
V_last = V_snaps[-1].reshape(4, 12)
np.set_printoptions(precision=2, suppress=True) # Pretty printing
```

<IPython.core.display.Image object>
Saved GIF to: ./mc_es_grid_evolution.gif

From the convergence snapshots and animation, several clear patterns emerge:

- Values near the terminal goal stabilize quickly. Because these states are visited frequently and have relatively deterministic returns, their value estimates settle early.
 - States far from the goal (particularly around the start) converge last. Their value estimates depend on long, noisy rollouts, so they require many more episodes to stabilize.
 - Cells next to the cliff settle more slowly at first due to policy oscillation between exploration and avoidance. However, actions that step directly into the cliff quickly become confidently poor: their Q-values plummet due to large negative returns and are effectively pruned by greedy policy updates.
-

1.5 Random walk on a ring: Why can monte-carlo method works? (Pen and Paper)

In this problem you will use a random walk problem to analyze the convergence behavior of Monte-Carlo control.

Consider a 1-D random walk problem on a ring, and one of them is the terminal state. Assume states are $\{0, 1, 2, \dots, K - 1\}$, and 0 is the terminal state. The transition model can be written as $x_{k+1} = \text{mod}(x_k + u_k)$ where $u_k = -1/1$.

Setup.

States $\mathcal{S} = \{0, 1, \dots, K - 1\}$ with terminal 0 (absorbing).

Actions $\mathcal{A} = \{-1, +1\}$, transition $s' = (s + a) \bmod K$ (Note that terminal is absorb, so once it arrive 0 it will not move).

Per-step reward -1 until termination; discount $\gamma \in (0, 1]$.

Distance to 0: $d(s) = \min\{s, K - s\}$.

Algorithm: first-visit Monte-Carlo control with **Exploring Starts**.

Claim. With probability 1, the learned policy converges to

$$\pi^*(s) = \begin{cases} \text{move to reduce } d(s), & d(s) \neq K/2, \\ \text{either action,} & d(s) = K/2 \text{ (if } K \text{ even),} \end{cases}$$

i.e., "go along the shortest arc to 0".

Proof (outward induction on d)

First we write down the optimal value V^* under π^*

(a) **TODO: write down V^***

Answer:

Here, under π^* the agent always moves one step toward 0 along a shortest arc.

Hence from any nonterminal state s , the hitting time to 0 is deterministic and equals

$$n(s) = d(s) = \min\{s, K - s\}$$

With per-step reward -1 until termination, the return from s is the length- $n(s)$ discounted sum

$$G(s) = \sum_{t=0}^{n(s)-1} \gamma^t (-1) = - \sum_{t=0}^{d(s)-1} \gamma^t$$

Therefore,

$$V^*(s) = - \sum_{t=0}^{d(s)-1} \gamma^t = \begin{cases} -\frac{1-\gamma^{d(s)}}{1-\gamma}, & \gamma \in (0, 1) \\ -d(s), & \gamma = 1 \end{cases} \quad \text{with } V^*(0) = 0$$

Base of induction ($d = 0$).

(b) TODO: Show that $Q(0, a) \equiv 0$

Answer:

Since 0 is the terminal state, the episode ends immediately upon reaching it. No further transitions occur, and no additional rewards are received. Therefore, for every action $a \in \mathcal{A}$, the return from $(0, a)$ is 0.

Hence,

$$Q(0, a) = 0, \quad \forall a \in \mathcal{A}$$

Inductive hypothesis. For every state with $d \leq m - 1$, the maximization of $Q(s, a)$ is already optimal, i.e.

$$\max_a Q(s, a) := V(s) = V^*$$

(c) TODO: For $d(s) = m$, show that $Q(s, \text{toward}) \geq Q(s, \text{away})$ (equality hold only when $s = \frac{K}{2}$)

Answer:

Suppose the current state s has distance $d(s) = m$ from the terminal 0.

If the agent takes the toward action, the next state s' will have distance $d(s') = m - 1$.

By the inductive hypothesis,

$$V(s') = V^*(s') = - \sum_{t=0}^{m-2} \gamma^t = -\frac{1-\gamma^{m-1}}{1-\gamma}$$

Thus the return is

$$Q(s, \text{toward}) = -1 + \gamma V(s') = -1 - \gamma \frac{1-\gamma^{m-1}}{1-\gamma} = -\frac{1-\gamma^m}{1-\gamma}$$

If the agent takes the away action, the next state s'' has distance $d(s'') = m + 1$ (unless $m = K/2$, in which case moving away keeps the same distance m).

By optimality of smaller distances,

$$Q(s, \text{away}) = -1 + \gamma V(s'') = -1 - \gamma \frac{1 - \gamma^{m+1}}{1 - \gamma} = -\frac{1 - \gamma^{m+1}}{1 - \gamma}$$

Since $\gamma \in (0, 1]$, we have

$$-\frac{1 - \gamma^m}{1 - \gamma} \geq -\frac{1 - \gamma^{m+1}}{1 - \gamma}$$

with equality only if $m = K/2$ (when the two arcs to 0 are symmetric).

Therefore,

$$Q(s, \text{toward}) \geq Q(s, \text{away})$$

and the greedy policy will choose the toward action, except in the symmetric case $d(s) = K/2$ where both actions are equivalent.

Therefore greedy improvement sets $\pi(s) = \text{toward}$. By convergence of monte-carlo evaluation under fix policy, $Q(s, \text{toward})$ finally converge.

1.6 Monte-Carlo control with ϵ -soft policies

TODO: finish the code for Monte-Carlo control with ϵ -soft policies

```
In [7]: def _epsilon_soft_from_Q(Q_row: np.ndarray, epsilon: float, tol: float = 1e-12) -> np.ndarray:
    """
    Build an  $\epsilon$ -soft *greedy* action distribution from a vector of Q-values.

    Rule:
    1) Give every action a base mass  $\epsilon/|A|$ .
    2) Split the remaining  $(1-\epsilon)$  uniformly among the greedy (argmax) actions.
       (Deterministic tie-handling: no random tie-breaking.)
    """
    nA = Q_row.shape[0]
    probs = np.full(nA, epsilon / nA, dtype=float)

    m = Q_row.max()
    ties = np.flatnonzero(np.isclose(Q_row, m, atol=tol, rtol=0.0))
    share = (1.0 - epsilon) / ties.size
    probs[ties] += share
    return probs

def mc_control_onpolicy_epsilon_soft(
    env,
    episodes: int = 5000,
    gamma: float = 0.95,
    max_steps: int = 500,
    alpha: float | None = None,      # None → sample mean 1/N(s,a); else constant step-size
    epsilon: float = 0.1,            # constant  $\epsilon$  if no schedule is provided
    eps_schedule=None,              # optional callable: eps = eps_schedule(ep) (GLIE-style)
    min_epsilon: float = 0.0,
):
    """
    On-policy first-visit Monte Carlo control with  $\epsilon$ -soft policies
    
```

(Sutton & Barto, On-policy MC Control).

Key differences from MC with Exploring Starts (ES):

- No exploring starts; episodes are generated by the current ϵ -soft policy π .
- Policy improvement *keeps* π ϵ -soft: greedy actions get $(1-\epsilon)+\epsilon/|A|$, others $\epsilon/|A|$.

"""

```
nS = env.observation_space.n
nA = env.action_space.n
```

```
Q = np.zeros((nS, nA), dtype=float)
N_first = np.zeros((nS, nA), dtype=int)
```

```
# Initialize  $\pi$  arbitrarily: uniform over actions (already  $\epsilon$ -soft for any  $\epsilon$ ).
policy = np.full((nS, nA), 1.0 / nA, dtype=float)
```

```
ep_idx, V_snaps = [], []
```

```
for ep in tqdm(range(episodes), desc="MC ( $\epsilon$ -soft)"):
    # GLIE-style  $\epsilon$  schedule (optional)
    eps = float(eps_schedule(ep)) if eps_schedule is not None else float(epsilon)
    eps = float(np.clip(eps, min_epsilon, 1.0))
```

```
# ---- Generate one episode under current  $\epsilon$ -soft policy  $\pi$  ----
```

```
states, actions, rewards = [], [], []
```

```
s, _ = env.reset()
```

```
a = int(np.random.choice(nA, p=policy[s])) # first action from current policy
```

```
for _ in range(max_steps):
```

```
#####
# TODO: sample action from policy and step in env
# hint: use env.step(a) to get (s', r, terminated, truncated, info)
```

```
# record state-action
states.append(s)
actions.append(a)
```

```
# step environment
```

```
s_next, r, terminated, truncated, _ = env.step(a)
rewards.append(r)
```

```
if terminated or truncated:
```

```
    break
```

```
# next action from current policy
```

```
a = int(np.random.choice(nA, p=policy[s_next]))
s = s_next
```

```
#####
# ---- First-visit markers for (s,a) within this episode ----
```

```
seen = set()
first_visit = [False] * len(states)
for t, (st, at) in enumerate(zip(states, actions)):
    if (st, at) not in seen:
        seen.add((st, at))
        first_visit[t] = True
```

```
# ---- Backward return accumulation; update Q only at first visits ----
```

```
#####
# ----
```

```

# TODO: update Q and N_first for first-visit states

G = 0.0
for t in reversed(range(len(states))):
    G = gamma * G + rewards[t]
    if first_visit[t]:
        st, at = states[t], actions[t]
        N_first[st, at] += 1
        if alpha is None:
            # sample-average step-size
            Q[st, at] += (G - Q[st, at]) / N_first[st, at]
        else:
            # constant step-size
            Q[st, at] += alpha * (G - Q[st, at])

#####
# ---- ε-soft greedy policy improvement (only on states seen this episode) ----
visited_states = set(states)
for st in visited_states:
    policy[st] = _epsilon_soft_from_Q(Q[st], eps)

# Light snapshots for convergence plots (every 100 eps)
if (ep % 100) == 0:
    V = Q.max(axis=1)
    V_snaps.append(V.copy())

return policy, Q, V_snaps

def eps_schedule(ep, c=2000.0, eps_min=0.01):
    """Simple GLIE schedule: εep = max(eps_min, c / (c + ep))."""
    return max(eps_min, c / (c + ep))

policy, Q, V_snaps = mc_control_onpolicy_epsilon_soft(
    env,
    episodes=10000,
    gamma=0.95,
    epsilon=1.0, # start fully exploratory; schedule will decay it
    eps_schedule=lambda ep: eps_schedule(ep, c=2000.0, eps_min=0.01),
    min_epsilon=0.0,
    alpha=0.01 # constant step-size for Q-updates (try None for 1/N(s,a))
)

V_soft = (policy * Q).sum(axis=1)
print_values(V_soft, 4, 12, "V from MC ε-soft policy")
print_policy(policy, 4, 12, "MC ε-soft derived policy")

```

MC (ε-soft): 100%|██████████| 10000/10000 [00:34<00:00, 293.65it/s]

```

V from MC  $\epsilon$ -soft policy
-11.89 -11.45 -10.91 -10.28 -9.64 -8.97 -8.21 -7.40 -6.53 -5.66 -4.70 -3.75
-12.40 -13.76 -14.16 -13.46 -13.24 -14.00 -12.68 -12.92 -10.67 -9.51 -4.56 -2.71
-13.36 -21.11 -30.08 -34.39 -42.39 -40.92 -43.48 -41.36 -32.00 -23.70 -9.69 -1.62
-18.73  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00

MC  $\epsilon$ -soft derived policy
> > > > > > > > > > > v
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ > v
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ > v
^ T T T T T T T T T T

```

Problem 2: SARSA & Q-learning on Cliffwalk environment

2.1 TD evaluation

Algorithmic Form. Suppose the agent is in state s_t , takes action $a_t \sim \pi(\cdot | s_t)$, receives reward r_t , and transitions to s_{t+1} . The TD(0) update rule is

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha [r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t)], \quad (2.6)$$

where $\alpha \in (0, 1]$ is the learning rate.

The term inside the brackets,

$$\delta_t = r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t), \quad (2.7)$$

is called the **TD error**.

TODO: finish the code for TD evalution

```
In [8]: def td0_evaluate(env, episodes=5000, gamma=0.95, alpha=0.1, seed=None):
    """
    On-policy temporal-difference evaluation under a **uniform random policy**.

    Computes:
    • V_TD0 : TD(0) / 1-step bootstrap   V(s_t) ← V(s_t) + α [ r_t + γ V(s_{t+1}) - V(s_t) ]
    • V_TD2 : 2-step TD for the *previous* state s_{t-1}
              V(s_{t-1}) ← V(s_{t-1}) + α [ r_{t-1} + γ r_t + γ^2 V(s_{t+1}) - V(s_{t-1}) ]
              (This is the n=2 special case of n-step TD.)
    Returns V_TD0 and RMSE traces for both estimators vs the provided V_random_gt.
    """

    nS, nA = env.observation_space.n, env.action_space.n
    policy = np.ones((nS, nA)) / nA

    V_TD0 = np.zeros(nS, dtype=float)
    V_TD2 = np.zeros(nS, dtype=float)

    visit_counts = np.zeros(nS, dtype=int) # diagnostic only
    errors_1, errors_2 = [], []

    if seed is not None:
        np.random.seed(seed)

    for _ in tqdm(range(episodes), desc="TD evaluation"):
```

```

s, _ = env.reset()
done = False

# Keep the previous transition to form a 2-step target
prev_state = None      # will hold  $s_{t-1}$ 
prev_reward = None     # will hold  $r_{t-1}$ 

while not done:
    #####
    # TODO: step the environment, update Q

    # sample action uniformly
    a = np.random.choice(nA, p=policy[s])
    s_next, r, terminated, truncated, _ = env.step(a)
    done = terminated or truncated

    # ---- TD(0) update ----
    td_target = r + (0 if done else gamma * V_TD0[s_next])
    V_TD0[s] += alpha * (td_target - V_TD0[s])

    # ---- 2-step TD update (for prev_state) ----
    if prev_state is not None:
        two_step_target = prev_reward + gamma * r
        if not done:
            two_step_target += (gamma ** 2) * V_TD2[s_next]
        V_TD2[prev_state] += alpha * (two_step_target - V_TD2[prev_state])

    # shift previous transition
    prev_state = s
    prev_reward = r
    s = s_next

    #####
    # Episode-Level RMSE diagnostics
    errors_1.append(np.sqrt(np.mean((V_TD0 - V_random_gt) ** 2)))
    errors_2.append(np.sqrt(np.mean((V_TD2 - V_random_gt) ** 2)))

return V_TD0, (errors_1, errors_2)

# Example run
V, (errors_1, errors_2) = td0_evaluate(env, episodes=1000, alpha=0.01)

print_values(V, 4, 12, "Estimated State Values (TD(0) evaluation)")

plt.figure()
plt.plot(errors_1, label="TD(0)")
plt.plot(errors_2, label="2-step TD")
plt.xlabel("Episode")
plt.ylabel("RMSE vs V_random_gt")
plt.grid(True)
plt.legend()
plt.show()

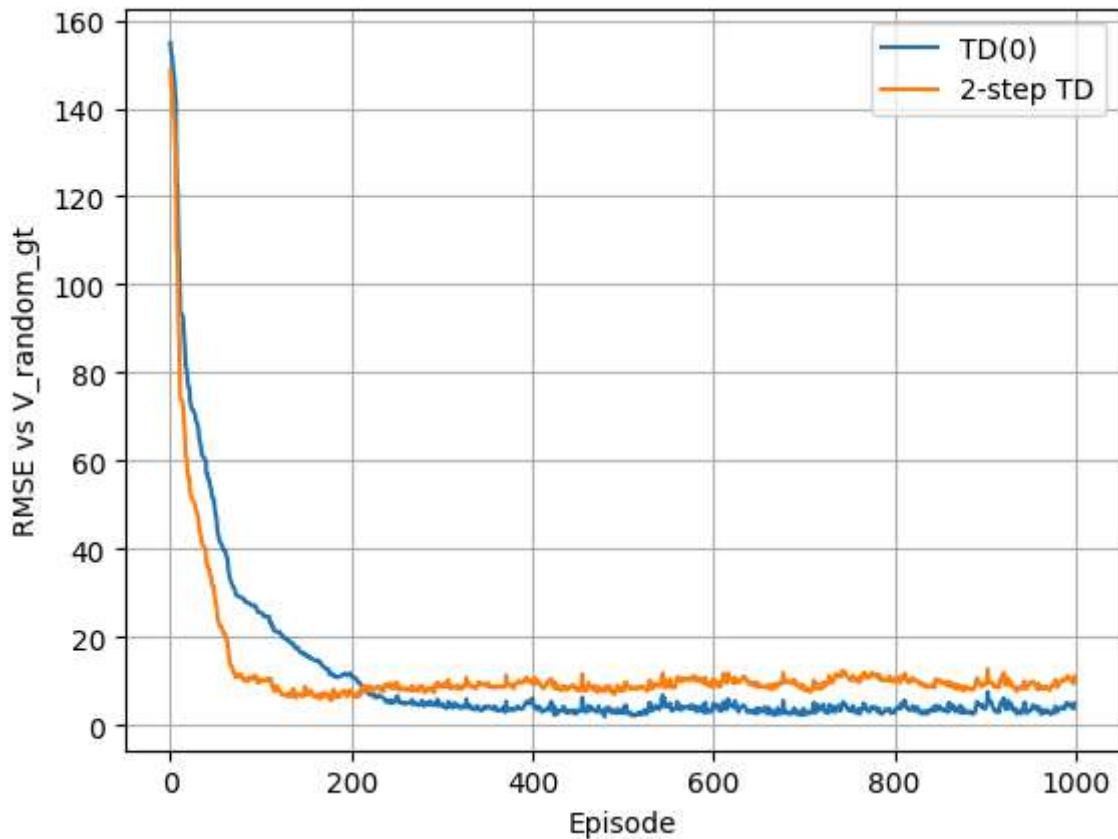
# Note: On my Laptop this may take ~20s for 1k episodes

```

TD evaluation: 100% |██████████| 1000/1000 [04:00<00:00, 4.16it/s]

Estimated State Values (TD(0) evaluation)

```
-144.70 -149.59 -152.56 -154.66 -157.15 -158.39 -157.10 -151.42 -143.12 -131.19 -120.00 -105.81  
-167.62 -173.36 -181.03 -183.70 -186.46 -190.49 -191.17 -186.31 -174.27 -159.71 -141.33 -109.89  
-206.84 -236.27 -242.34 -243.95 -252.48 -258.11 -261.00 -259.76 -246.50 -231.48 -218.37 -105.63  
-250.13  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
```



2.2 SARSA algorithm

In [9]:

```
def sarsa(  
    env,  
    episodes=5000,  
    Q_init=None,  
    gamma=0.95,  
    epsilon=0.01,  
    alpha=0.1,  
    max_steps=500,      # per-episode safety cap  
):  
    """  
    On-policy SARSA(0) control for **tabular, discrete** Gymnasium environments.  
  
    Behavior policy:  $\epsilon$ -greedy w.r.t. current  $Q$ .  
    Update rule (per step):  
         $Q(s,a) \leftarrow Q(s,a) + \alpha [ r + \gamma Q(s', a') - Q(s,a) ]$ ,  
    where  $a'$  is sampled  $\epsilon$ -greedily from state  $s'$  (on-policy).  
    """  
    nS, nA = env.observation_space.n, env.action_space.n  
    Q = Q_init.copy() if Q_init is not None else np.zeros((nS, nA), dtype=float)  
    visits = np.zeros((nS, nA), dtype=int)  
    V_snaps = []  
  
    def eps_greedy_action(s, eps):  
        """ $\epsilon$ -greedy over  $Q[s]$ : uniform random with prob  $\epsilon$ ; otherwise greedy with random tie-break
```

```

    if np.random.rand() < eps:
        return np.random.randint(nA)
    row = Q[s]
    m = row.max()
    candidates = np.flatnonzero(row == m)
    return int(np.random.choice(candidates))

for ep in tqdm(range(episodes), desc="SARSA control"):
    s, _ = env.reset()
    a = eps_greedy_action(s, epsilon)

    # (Optional) record of visited states; useful for debugging/plots.
    s_record = []

    for _ in range(max_steps):
        #####
        # TODO: step the environment, update Q and choose action

        s_record.append(s)

        # step the environment
        s_next, r, terminated, truncated, _ = env.step(a)
        done = terminated or truncated

        # pick next action on-policy
        if not done:
            a_next = eps_greedy_action(s_next, epsilon)
        else:
            a_next = None

        # update Q(s,a)
        td_target = r + (0 if done else gamma * Q[s_next, a_next])
        Q[s, a] += alpha * (td_target - Q[s, a])

        visits[s, a] += 1

        if done:
            break

        # shift to next transition
        s, a = s_next, a_next

        #####
        # Lightweight snapshots early on (every 2 episodes up to 100)
        if (ep % 2) == 0 and ep <= 100:
            V_snaps.append(Q.max(axis=1).copy())

return Q, visits, V_snaps

```

Q, visits, V_snaps = sarsa(env, episodes=50000, gamma=0.95, alpha=0.01, epsilon=0.01)
V = Q.max(axis=1)
print_values(V, 4, 12, title="State Values (greedy after SARSA)")

SARSA control: 100%|██████████| 50000/50000 [00:33<00:00, 1486.61it/s]

State Values (greedy after SARSA)

-9.49	-9.13	-8.69	-8.19	-7.67	-7.10	-6.50	-5.86	-5.17	-4.45	-3.68	-2.86
-9.77	-9.23	-8.67	-8.07	-7.44	-6.77	-6.06	-5.32	-4.55	-3.73	-2.87	-1.95
-10.28	-9.74	-9.18	-8.58	-8.04	-6.62	-5.49	-4.63	-3.85	-2.96	-1.96	-1.00
-10.78	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

2.3 Q-learning algorithm

In [10]:

```
def q_learning(
    env,
    episodes=20000,
    gamma=0.95,
    epsilon=0.01,
    alpha=0.1,
    max_steps=500,      # per-episode safety cap
):
    """
    Tabular **Q-learning** (off-policy, 1-step TD) control.

    Behavior policy: ε-greedy w.r.t. current Q (for exploration).
    Target policy:   greedy (max over actions in s') – this is what makes it **off-policy**.

    Update (per step):
        Q(s,a) ← Q(s,a) + α [ r + γ * max_{a'} Q(s', a') - Q(s,a) ].
    """

    nS, nA = env.observation_space.n, env.action_space.n
    Q = np.zeros((nS, nA), dtype=float)
    visits = np.zeros((nS, nA), dtype=int) # diagnostic: how often each (s,a) is updated
    V_snaps = []

    def eps_greedy_action(s, eps):
        """ε-greedy over Q[s]: with prob ε pick random action; else greedy with random tie-break
        if np.random.rand() < eps:
            return np.random.randint(nA)
        row = Q[s]
        m = row.max()
        candidates = np.flatnonzero(row == m) # random tie-breaking argmax
        return int(np.random.choice(candidates))"""

    for ep in tqdm(range(episodes), desc="Q-learning control"):
        s, _ = env.reset()

        for _ in range(max_steps):
            #####
            # TODO: step the environment, update Q and choose action

            # choose action from ε-greedy policy (behavior)
            a = eps_greedy_action(s, epsilon)

            # step environment
            s_next, r, terminated, _ = env.step(a)
            done = terminated or truncated

            # Q-Learning update (target uses greedy a')
            td_target = r + (0 if done else gamma * np.max(Q[s_next]))
            Q[s, a] += alpha * (td_target - Q[s, a])
```

```
visits[s, a] += 1

if done:
    break

# move to next state
s = s_next

#####
# Light snapshots early on (every 10 episodes up to 1k)
if (ep % 10) == 0 and ep <= 1000:
    V_snaps.append(Q.max(axis=1).copy())

return Q, V_snaps

Q, V_snaps = q_learning(env, episodes=50000, gamma=0.95, alpha=0.01, epsilon=0.01)
V = Q.max(axis=1)
print_values(V, 4, 12, title="State Values (greedy after Q-learning)")
```

Q-learning control: 100% | ██████████ | 50000/50000 [00:35<00:00, 1410.06it/s]
State Values (greedy after Q-learning)
-8.72 -8.44 -8.06 -7.61 -7.11 -6.58 -6.01 -5.40 -4.77 -4.11 -3.43 -2.77
-8.98 -8.63 -8.19 -7.69 -7.15 -6.55 -5.91 -5.22 -4.48 -3.69 -2.85 -1.95
-9.19 -8.62 -8.03 -7.40 -6.73 -6.03 -5.30 -4.52 -3.71 -2.85 -1.95 -1.00
-9.73 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00