COMP3667: Reinforcement learning practical 4

Dynamic Programming

robert.lieck@durham.ac.uk & christopher.g.willcocks@durham.ac.uk

This is the version *with* answers!

1 Overview

Welcome to the fourth reinforcement learning practical. In this practical, we will dive a bit more into the practical details of dynamic programming methods. In particular, we will:

- Use the OpenAI Gym "Frozen Lake" environment as a basic example.
- Implement *policy evaluation* and understand how this algorithm updates state values.
- Implement *policy improvement* to learn better policies.
- Implement *policy iteration* to learn optimal policies.

For this practical, you will need a basic Python environment with `numpy`, `matplotlib`, and OpenAI `gym` (version 0.20.0).

You can download the Jupyter notebook **here** or directly open the **Colab notebook**.

2 Frozen Lake Environment

Install OpenAI `gym` if you have not done this already. You will need to use version 0.20.0: </> copy code

```
%%capture
!pip install 'gym[box2d]==0.20.0'
```

Do the relevant imports </> copy code

```
import gym
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.font_manager
```

and load the Frozen Lake environment </> copy code

```
name = 'FrozenLake-v1'  # small version
# name = 'FrozenLake8x8-v1'  # larger version
env = gym.make(name, is_slippery=False)
env.seed(742)
env.action_space.seed(742)
# named actions
LEFT, DOWN, RIGHT, UP = 0, 1, 2, 3
```

The code of the environment is at `https://github.com/openai/gym/blob/master/gym/envs/toy_text/frozen_lake.py`. Use `slippery=False` for now, because otherwise the optimal solutions are not very intuitive (with `slippery=True`, when the agent tries to move forward it will with equal probability instead slip sideways, but never backwards; therefore it is sometimes better to try to move "away from a hole" instead of "towards the goal"). Use this helper function to visualise the environment in a nice way: </> copy code

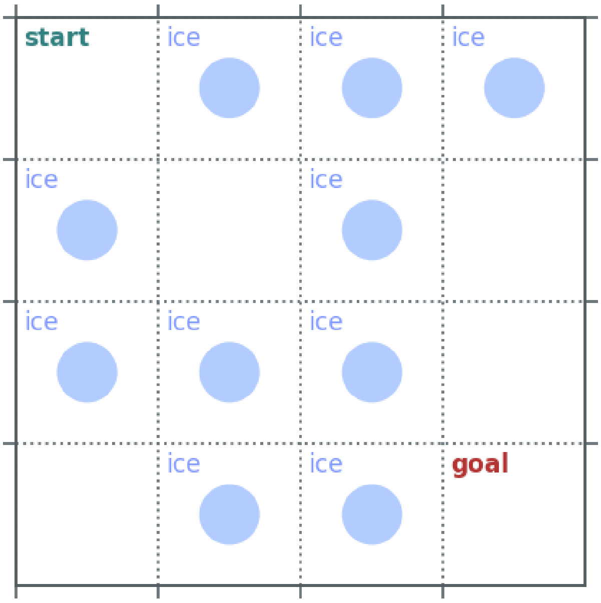
```
def plot(env, v=None, policy=None, col_ramp=1, dpi=175, draw_vals=False, mark_ice=True):
    # set up plot
    plt.rcParams['figure.dpi'] = dpi
    plt.rcParams.update({'axes.edgecolor': (0.32,0.36,0.38)})
    plt.rcParams.update({'font.size': 4 if env.env.nrow == 8 else 7})
    gray = np.array((0.32,0.36,0.38))
    plt.figure(figsize=(3, 3))
    ax = plt.gca()
    ax.set_xticks(np.arange(env.env.ncol)-.5)
    ax.set_yticks(np.arange(env.env.nrow)-.5)
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    plt.grid(color=(0.42,0.46,0.48), linestyle=':')
    ax.set_axisbelow(True)
    ax.tick_params(color=(0.42,0.46,0.48),
                  which='both', top='off', left='off', right='off', bottom='off')
    # use zero value as dummy if not provided
    if v is None:
        v = np.zeros(env.nS)
    # plot values
    plt.imshow(1-v.reshape(env.env.nrow,env.env.ncol)**col_ramp,
              cmap='gray', interpolation='none',
              clim=(0,1), zorder=-1)
    # go through states
    for s in range(env.nS):
        x, y = s % env.env.nrow, s // env.env.ncol
        # print numeric values
        if draw_vals and v[s] > 0:
            vstr = '{0:.1e}'.format(v[s]) if env.env.nrow == 8 else '{0:.6f}'.format(v[s])
            plt.text(x - 0.45, y + 0.45, vstr, color=(0.2, 0.8, 0.2), fontname='Sans')
        # mark ice, start, goal
        if env.desc.tolist()[y][x] == b'F':
            plt.text(x-0.45,y-0.3, 'ice', color=(0.5, 0.6, 1), fontname='Sans')
            if mark_ice:
                ax.add_patch(plt.Circle((x, y), 0.2, color=(0.7, 0.8, 1), zorder=0))
        elif env.desc.tolist()[y][x] == b'S':
            plt.text(x-0.45,y-0.3, 'start',color=(0.2,0.5,0.5), fontname='Sans',
                     weight='bold')
        elif env.desc.tolist()[y][x] == b'G':
            plt.text(x-0.45,y-0.3, 'goal', color=(0.7,0.2,0.2), fontname='Sans',
                     weight='bold')
            continue # don't plot policy for goal state
        else:
            continue # don't plot policy for holes
        # plot policy
        def plot_arrow(x, y, dx, dy, v, scale=0.4):
            plt.arrow(x, y, scale * float(dx), scale * float(dy), color=gray+0.2*(1-v),
                      head_width=0.1, head_length=0.1, zorder=1)
        if policy is not None:
            a = policy[s]
            if a[0] > 0.0: plot_arrow(x, y, -a[0],    0., v[s]) # left
            if a[1] > 0.0: plot_arrow(x, y,    0.,  a[1], v[s]) # down
            if a[2] > 0.0: plot_arrow(x, y,  a[2],    0., v[s]) # right
            if a[3] > 0.0: plot_arrow(x, y,    0., -a[3], v[s]) # up
    plt.show()
```

Have a look

</> copy code

```
print('action space: ' + str(env.action_space))
print('reward range: ' + str(env.reward_range))
print('observation space: ' + str(env.observation_space))
```

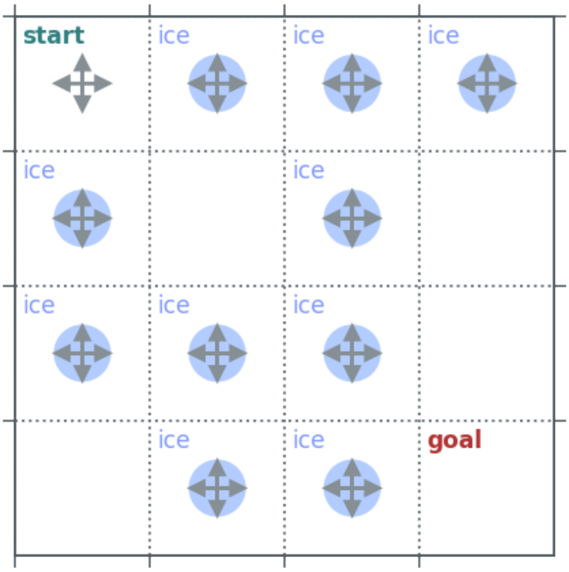
```
plot(env=env)
--> action space: Discrete(4)
--> reward range: (0, 1)
--> observation space: Discrete(16)
```



- Define a uniform policy and plot it in the environment. *Hint:* `env.nS` and `env.nA` give you the number of states and actions, respectively; you can provide the policy to the plotting function via the `policy` argument.

  Answer: </> copy code

```
def uniform_policy(env):
    return np.ones((env.nS, env.nA)) / env.nA
plot(env=env, policy=uniform_policy(env))
```



3 Policy Evaluation

Now, we would like to know how well a policy performs, that is, what the state value of a particular state is when following the policy in the future. This *policy evaluation* procedure can be efficiently done using dynamic programming, which iteratively improves state value estimates and converges to the true state values.

- Write a `policy_eval_step` function that takes an initial state value estimate $v_\pi^k(s)$ and computes an improved estimate $v_\pi^{k+1}(s)$ as

$$v_\pi^{k+1}(s) = \sum_a \pi(a \mid s) \sum_{s'} p(s' \mid s, a) \left[\mathcal{R}_{ss'}^a + \gamma \, v_\pi^k(s')\right] \ . \tag{1}$$

  Hint: `env.env.P[s][a]:` is giving you a list of tuples `(p, s', r, done)`, one for each possible transition from state `s` when taking action `a`: `p` is the probability of this transition to happen, `s'` is the state the agent transitions to, `r` is the reward it receives, and `done` indicates whether the episode is over (which you will not need to use).

  Answer: </> copy code

```python
def policy_eval_step(env, policy, gamma, v_init=None):
    if v_init is None:
        v_init = np.zeros(env.nS)
    v = np.zeros(env.nS)
    for s_from in range(env.nS):
        for a in range(env.nA):
            pi = policy[s_from, a]
            for p, s_to, r, done in env.env.P[s_from][a]:
                v[s_from] += pi * p * (r + gamma * v_init[s_to])
    return v
```

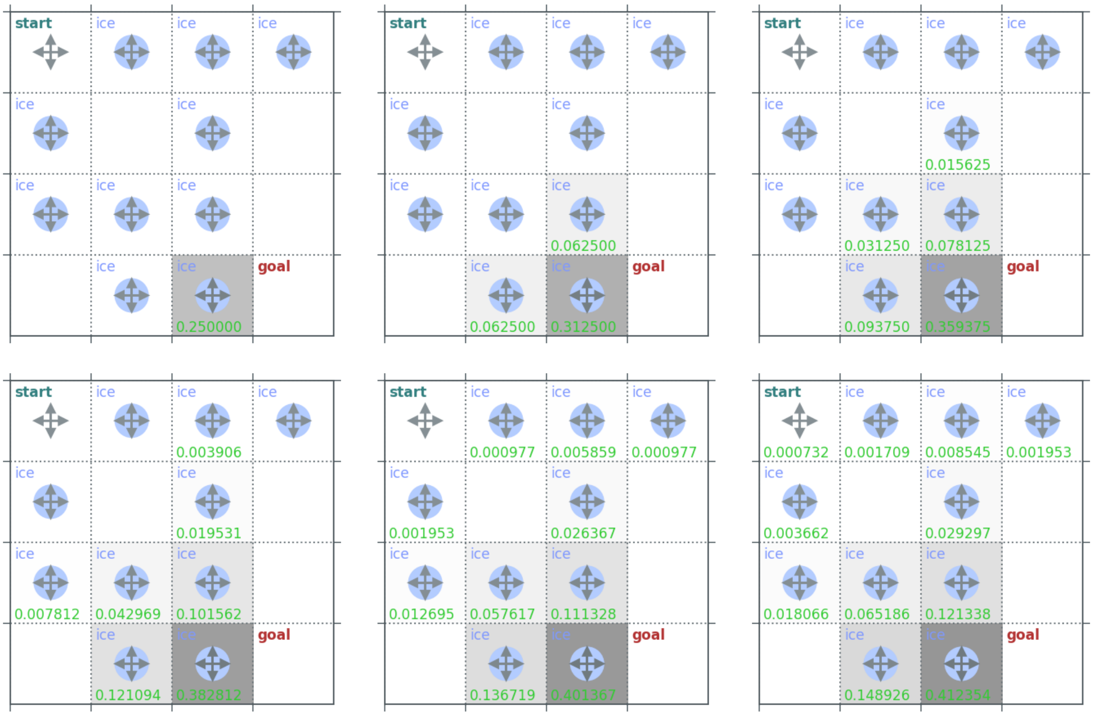
- Initialise the state values with zero, take one `policy_eval_step` at a time and plot the result to observe how updates are being performed. *Hint:* You can provide the state values to the plot function using the `v` argument, specifying `draw_vals=True` additionally shows the numeric state values.

  First, initialise the state values: </> copy code

```python
v = np.zeros(env.nS)
```

  Then repeatedly execute the update step: </> copy code

```python
v = policy_eval_step(env, uniform_policy(env), 1, v)
plot(env, v, uniform_policy(env), draw_vals=True)
```



- On an intuitive level, how would you describe the dynamics you see? What seems to be inefficient about the current implementation? How could this be improved?

  The environment is “flooded” with values from the goal state (or, more generally, any reward transition); values “propagate” one step with each iteration. In the first iterations, most state values do not change (they remain zero) and would not need to be updated, but the current implementation always updates *all* state values, which seems inefficient. If we knew which state values change, we could only update those.

- Implement a modified `policy_eval_step_inplace` version of the function that performs state value updates in-place. That is, instead of clearly separating the “old” values $v_\pi^k(s)$ and the “new” values $v_\pi^{k+1}(s)$, you operate on a single state value estimate $v_\pi(s)$, which is updated as you go. *Hint:* Make sure the updates for a particular state are still “atomic” and you do not use the half-computed values (in case of transitions that *stay* in a particular state).

  - Think about a clever order in which to update state values in-place. *Hint:* States are ordered from top-left (start: 0) to bottom-right (goal: 15).

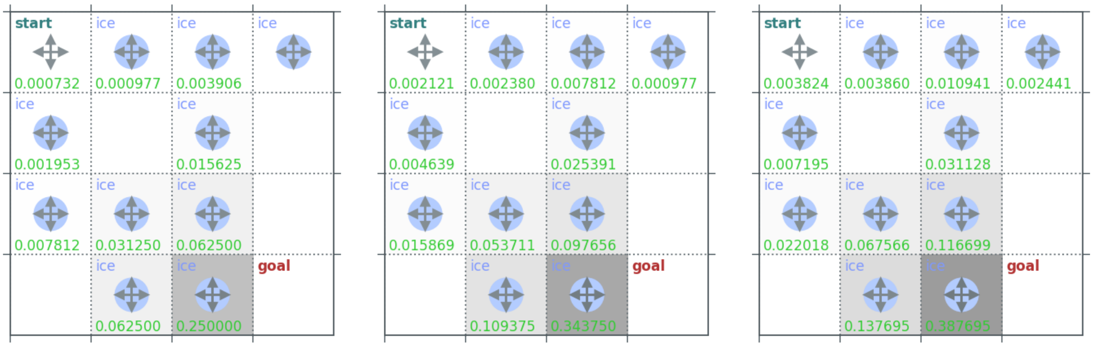
    In the modified implementation, the order of states is reversed because this makes updates more efficient as newly computed values will directly be reused in the following updates (the update direction is the same as the “flow” of state values from goal state to start state; this is obviously specific to this environment and not a general recipe).

  - Again, observe the step-wise update of values from one iteration to the next. How does that compare to the original implementation?

    Already in the first iteration, values propagate all the way to the start state.

  Answer: </> copy code

```
def policy_eval_step_inplace(env, policy, gamma, v_init=None):
    if v_init is None:
        v_init = np.zeros(env.nS)
    v = v_init.copy() # opearate on copy in-place
    for s_from in reversed(range(env.nS)):  # reverse order of states
        v_s_from = 0  # compute value for this state
        for a in range(env.nA):
            pi = policy[s_from, a]
            for p, s_to, r, done in env.env.P[s_from][a]:
                v_s_from += pi * p * (r + gamma * v[s_to])  # use the values we also update
        v[s_from] = v_s_from  # update
    return v
```



- Write a `policy_evaluation` function that interatively updates the state values using the `policy_eval_step` or `policy_eval_step_inplace` function and stops if they do not change (by some small tolerance value). Print the number of iterations needed to converge and compare for the `policy_eval_step` and `policy_eval_step_inplace` implementation.

  Answer: </> copy code

```
def policy_evaluation(env, policy, gamma, v_init=None,
                      print_iter=False, atol=1e-8, max_iter=10**10):
    if v_init is None:
        v_init = np.zeros(env.nS)
    v = v_init
    for i in range(1, max_iter + 1):
        new_v = policy_eval_step(env, policy, gamma, v)
        if np.allclose(v, new_v, atol=atol):
            break
        v = new_v
    if print_iter:
        print(f"{i} iterations")
    return v
```

- How many iterations do you need until state values have converged to their true value? To make it simpler, do the following though experiment: Take an environment that has only a single state and a single action (so nothing can really change and there is only one possible policy) and you get a reward of 1 upon every transition. Take the update equation for the state value from above, which now simplifies to

  $$v^{k+1} = 1 + \gamma\, v^{k}\ . \tag{2}$$

  Can you write down v^n in a non-recursive form assuming you start with $v^0 = 0$? Can you write down v^∞ in closed form? Is v^∞ the exact state value? How long does it take to converge? What happens for $\gamma = 1$ as opposed to $\gamma < 1$?

  Starting with $v^0 = 0$, v^n can be "unrolled" and rewritten as

  $$\begin{aligned}
  v^n &= 1 + \gamma\, v^{n-1} \tag{3}\\
  &= 1 + \gamma\,(1 + \gamma\, v^{n-2}) \tag{4}\\
  &= \cdots \tag{5}\\
  &= 1 + \gamma\,(1 + \gamma \cdots (1 + \gamma\, v^0)) \tag{6}\\
  &= 1 + \gamma + \gamma^2 \cdots (1 + \gamma^2\, v^0) \tag{7}\\
  &= 1 + \gamma + \gamma^2 + \cdots + \gamma^{n-1} + \gamma^n\, v^0 \tag{8}\\
  &= \sum_{k=0}^{n-1} \gamma^k \qquad \text{(non-recursive; geometric series)} \tag{9}\\
  &= \frac{1-\gamma^n}{1-\gamma} \qquad \text{(closed form formula for } \gamma < 1\text{)} \tag{10}\\
  \Rightarrow \qquad v^\infty &= \frac{1}{1-\gamma} \tag{11}
  \end{aligned}$$

  Inserting v^∞ into the update equation gives

  $$\begin{aligned}
  \frac{1}{1-\gamma} &= 1 + \gamma \frac{1}{1-\gamma} \tag{12}\\
  \Leftrightarrow \qquad 1 &= 1 - \gamma + \gamma \tag{13}\\
  \Leftrightarrow \qquad 1 &= 1 \tag{14}\\
  & \qquad \square \tag{15}\\
  & \tag{16}
  \end{aligned}$$

  so v^∞ is indeed the exact state value. The values converge exponentially fast (the γ^n term decays at an exponential rate), but will never be exact if there are paths of infinite length in the environment (typically loops; like the self-loop in this minimal though experiment). If only finite-length paths exist, the series can be truncated and values will be exact after a finite number of iterations.

4 Policy Improvement

- Implement a function that computes state-action values $q_\pi(s, a)$ (for all actions in a given state) from the state values $v_\pi(s)$ using their known relation

  $$q_\pi(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma\, v_\pi(s')\right]\ . \tag{17}$$

  Answer: </> copy code

```
def q_from_v(env, v, s, gamma):
    q = np.zeros(env.nA)
    for a in range(env.nA):
        for p, s_to, r, done in env.P[s][a]:
            q[a] += p * (r + gamma * v[s_to])
    return q
```

- Implement a policy_improvement function that takes a state value estimate and defines a policy (by first computing state-action values) that achieves maximal return (i.e. always chooses an action with maximal state-action value). Optionally, either choose actions deterministically, or choose all actions with maximum value with the same probability (if there is more than one such action).

  Answer: </> copy code

```
def policy_improvement(env, v, gamma, deterministic=False):
    policy = np.zeros([env.nS, env.nA]) / env.nA
    for s in range(env.nS):
        q = q_from_v(env, v, s, gamma)
        if deterministic:
            # deterministic policy
            policy[s][np.argmax(q)] = 1
        else:
            # stochastic policy with equal probability on maximizing actions
            best_a = np.argwhere(q==np.max(q)).flatten()
            policy[s, best_a] = 1 / len(best_a)
    return policy
```

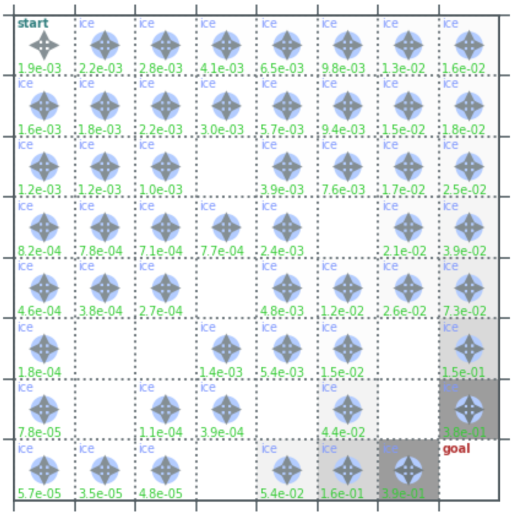
- Load the larger 'FrozenLake8x8-v1' environment (again with is_slippery=False for now), compute state values by evaluating the uniform policy, plot the result. Then compute an improved policy and plot the result again.

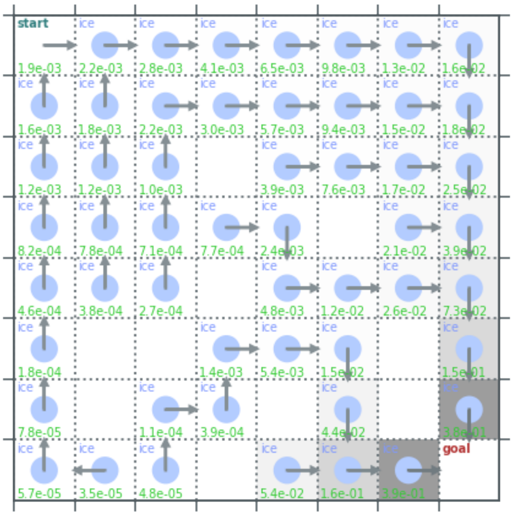
  Load and plot the larger environment with uniform policy: </> copy code

```
env = gym.make('FrozenLake8x8-v1', is_slippery=False)
env.seed(742)
env.action_space.seed(742)
policy = uniform_policy(env)
v = policy_evaluation(env, policy, gamma=1)
plot(env, v, policy, draw_vals=True)
```

Compute improved policy and plot again: </> copy code

```
new_policy = policy_improvement(env, v, gamma=1)
plot(env, v, new_policy, draw_vals=True)
```


start
ice
1.9e-03 2.2e-03 2.8e-03 4.1e-03 6.5e-03 9.8e-03 1.3e-02 1.6e-02
1.6e-03 1.8e-03 2.2e-03 3.0e-03 5.7e-03 9.4e-03 1.5e-02 1.8e-02
1.2e-03 1.2e-03 1.0e-03 3.9e-03 7.6e-03 1.7e-02 2.5e-02
8.2e-04 7.8e-04 7.1e-04 7.7e-04 2.4e-03 2.1e-02 3.9e-02
4.6e-04 3.8e-04 2.7e-04 4.8e-03 1.2e-02 2.6e-02 7.3e-02
1.8e-04 1.4e-03 5.4e-03 1.5e-02 1.5e-01
7.8e-05 1.1e-04 3.9e-04 4.4e-02 3.8e-01
5.7e-05 3.5e-05 4.8e-05 5.4e-02 1.6e-01 3.9e-01
goal


start
ice
1.9e-03 2.2e-03 2.8e-03 4.1e-03 6.5e-03 9.8e-03 1.3e-02 1.6e-02
1.6e-03 1.8e-03 2.2e-03 3.0e-03 5.7e-03 9.4e-03 1.5e-02 1.8e-02
1.2e-03 1.2e-03 1.0e-03 3.9e-03 7.6e-03 1.7e-02 2.5e-02
8.2e-04 7.8e-04 7.1e-04 7.7e-04 2.4e-03 2.1e-02 3.9e-02
4.6e-04 3.8e-04 2.7e-04 4.8e-03 1.2e-02 2.6e-02 7.3e-02
1.8e-04 1.4e-03 5.4e-03 1.5e-02 1.5e-01
7.8e-05 1.1e-04 3.9e-04 4.4e-02 3.8e-01
5.7e-05 3.5e-05 4.8e-05 5.4e-02 1.6e-01 3.9e-01
goal

5 Policy Iteration

- Starting from the improved policy from above, perform two updates by doing
  - policy evaluation (plot the results)
  - policy improvement (plot the results).

  Use a stochastic policy improvement (i.e. choose optimal action with equal probability) and a discount value of $\gamma = 1$. What do you observe? To you see anything that could be problematic? Can you explain what you observe (you may need to print the raw state values)? Would a deterministic policy improvement help? *Hint:* Remember what we learned above about value convergence and paths of finite/infinite length.

  First update: The improved policy is deterministic and during evaluation state values of 1 propagate through the entire state space. In the policy improvement step we get an almost uniform policy (except it does not jump into holes) because state values are equal everywhere.

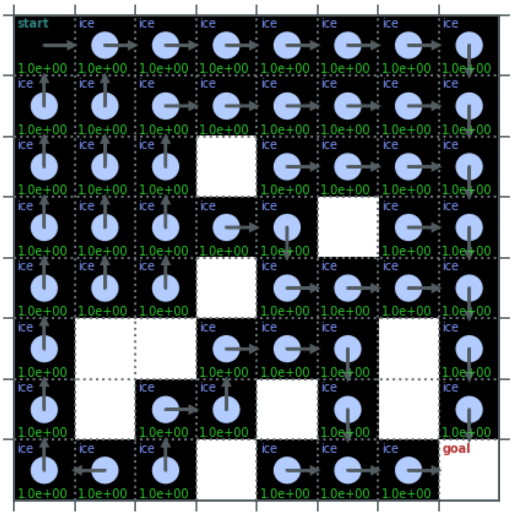
  Second update: The state values from the (almost) uniform policy do not converge to their exact value of 1, because we have infinite-length paths (loops). Therefore, in the policy improvement step, we again get a deterministic policy (moving towards the goal state, because these state values have "seen" the reward in earlier iterations and are therefore converged more closely to the exact value of 1).

  Potential problems: We do not get a stable policy, but instead alternate between the deterministic and stochastic version with state values being exactly 1 in the deterministic case and only approximately 1 in the stochastic case. This can be problematic when trying to detect convergence (as both policy and values keep on changing).

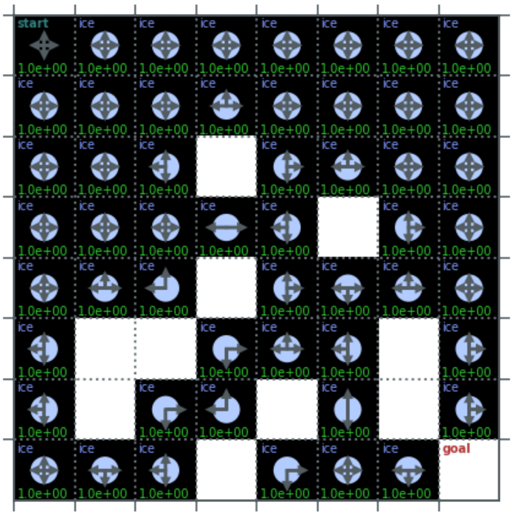
  Deterministic policy: A deterministic policy improvement step **cannot** solve this problem (we may still get loops if we are unlucky). In the contrary, since all state values are the same, the deterministic policy may choose arbitrary actions (as long as they do not jump into a hole), which in total may lead to an arbitrarily bad policy.

</> copy code

```
gamma = 1
v = policy_evaluation(env, new_policy, gamma=gamma)
plot(env, v=v, policy=new_policy, draw_vals=True)
print(v)
new_policy = policy_improvement(env, v, gamma=gamma)
plot(env, v=v, policy=new_policy, draw_vals=True)
```



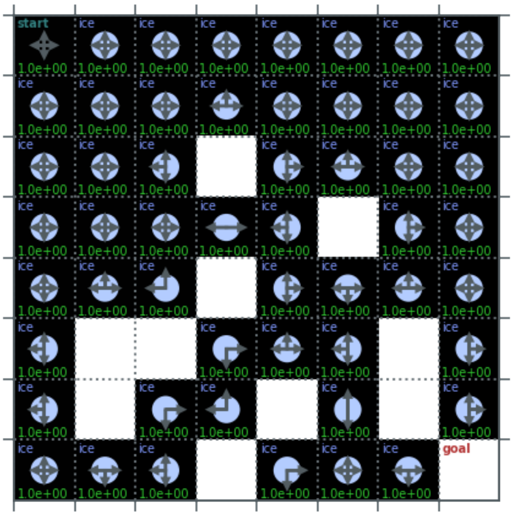
Policy Evaluation #1



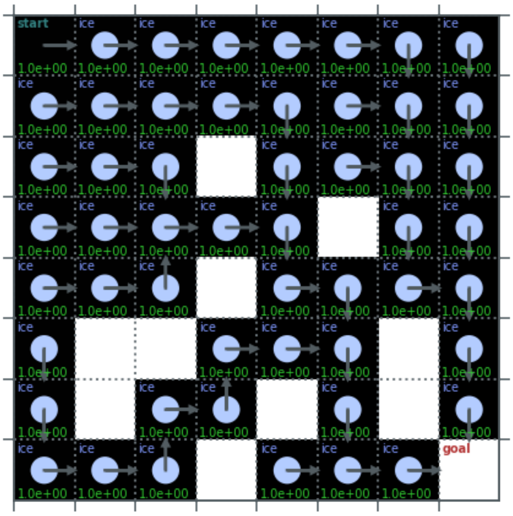
Policy Improvement #1

State values:

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 0. 0. 1. 1. 1. 0. 1.
 1. 0. 1. 1. 0. 1. 0. 1. 1. 1. 1. 0. 1. 1. 1. 0.]
```



Policy Evaluation #2



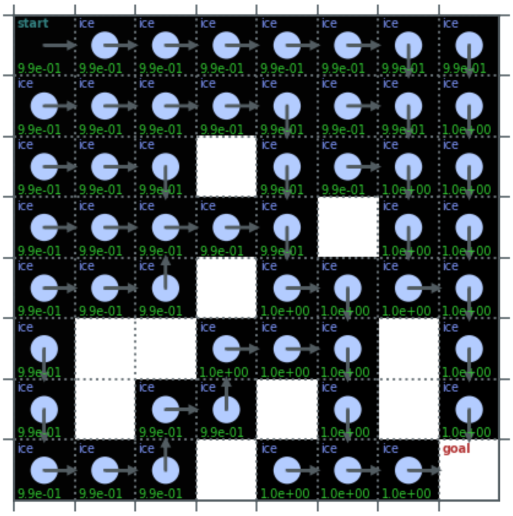
Policy Improvement #2

State values:

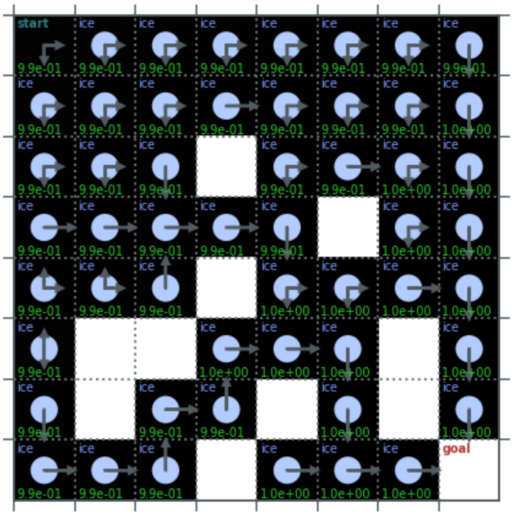
```
[0.99584585 0.99588645 0.99596896 0.99609428 0.99624201 0.996371
 0.99647678 0.99653576 0.99584504 0.99588397 0.99596476 0.99610928
 0.99629677 0.99642897 0.99655735 0.99662794 0.99584513 0.99587906
 0.99593551 0.         0.9964423  0.99652499 0.99672868 0.99682301
 0.99585109 0.99589112 0.99599191 0.99625233 0.9965307  0.
 0.9970407  0.99714287 0.99585676 0.9958818  0.9959271  0.
 0.9969224  0.99720212 0.99727183 0.99759226 0.99587711 0.
 0.         0.99683596 0.99705648 0.99743225 0.         0.99838516
 0.99592708 0.         0.99644137 0.99663059 0.         0.9980566
 0.         0.99918967 0.99600632 0.99612383 0.9962692  0.
 0.99868395 0.99869025 0.99934277 0.        ]
```

- Change the value to $\gamma = 0.999$ and do another two updates. What is different? *Hint:* You can use $\gamma = 0.9$ to see the effect more clearly.

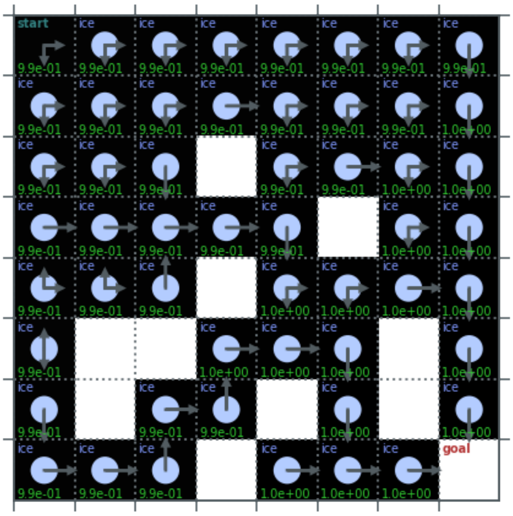
  State values are almost the same, but a value of $\gamma < 1$ penalises moving in circles "for no reason". Therefore, we get a policy the is moving towards the goal state as quickly as possible without creating any loop (even though there are sometimes multiple equally-valued paths, so the policy is not deterministic). As a result we obtain a stable policy and (as a consequence) stable state values.


start
ice
goal
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 1.0e+00
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00 1.0e+00 1.0e+00
9.9e-01 1.0e+00 1.0e+00 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00 1.0e+00

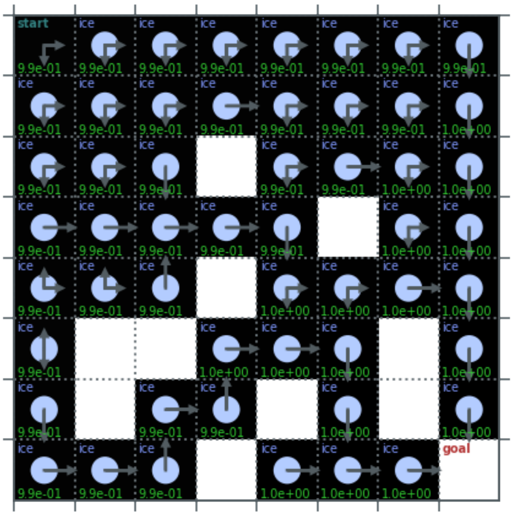
Policy Evaluation #3 ($\gamma = 0.999$)


start
ice
goal
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 1.0e+00
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00 1.0e+00 1.0e+00
9.9e-01 1.0e+00 1.0e+00 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00 1.0e+00

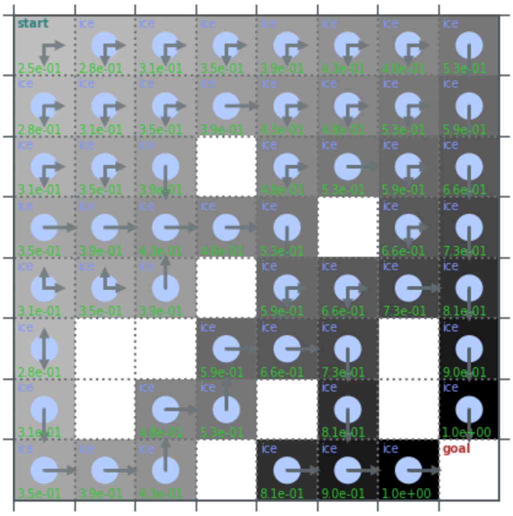
Policy Improvement #3 ($\gamma = 0.999$)


start
ice
goal
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 1.0e+00
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00 1.0e+00 1.0e+00
9.9e-01 1.0e+00 1.0e+00 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00 1.0e+00

Policy Evaluation #4 ($\gamma = 0.999$)


start
ice
goal
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 1.0e+00
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00 1.0e+00 1.0e+00
9.9e-01 1.0e+00 1.0e+00 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00
9.9e-01 9.9e-01 9.9e-01 1.0e+00 1.0e+00 1.0e+00

Policy Improvement #4 ($\gamma = 0.999$)



Converged Policy/Values ($\gamma = 0.9$)