2.강
Dynamic Programming
(Part 1)

Contents

- MDP review
- Dynamic Programming (DP)
- Dynamic Programming for a control problem
- Value Iteration
- Code exercise

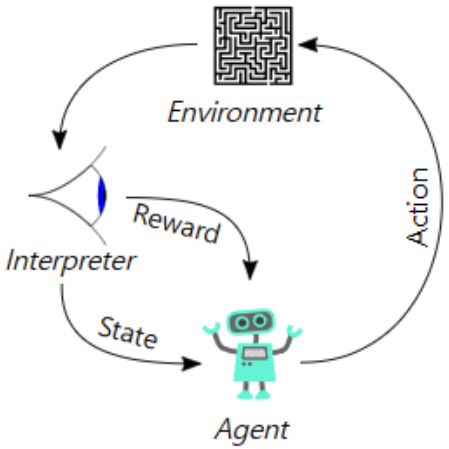
MDP Code Review

Quick view of the Gym library

Gym is a library for reinforcement learning research. It provides us with a simple interface to a large number of tasks, including

- Classic control tasks (CartPole, Pendulum, MountainCar, etc)
- Classic video games (Space Invaders, Breakout, Pong, etc)
- Continuous control tasks
- Robotic arm manipulation

In this section we are going to get familiar with the five methods that we'll use while solving a control task.



Maze env.: Find the exit

States and state space

```
env = Maze()
```

States and state space

The states consist of a tuple of two integers, both in the range [0, 4], representing the row and column in which the agent is currently located:

$$s = (row, column)\ row, column \in \{0, 1, 2, 3, 4\}$$

The state space (set of all possible states in the task) has 25 elements (all possible combinations of rows and columns):

$$Rows \times Columns\ S = \{(0,0), (0,1), (1,0), \dots\}$$

Information about the state space is stored in the env.observation_space property. In this environment, it is of MultiDiscrete([5 5]) type, which means that it consists of two elements (rows and columns), each with 5 different values.

```
print(f"For example, the initial state is: {env.reset()}")
print(f"The space state is of type: {env.observation_space}")
```

Action space

Actions and action space

In this environment, there are four different actions and they are represented by integers:

$$a \in \{0, 1, 2, 3\}$$

- 0 -> move up
- 1 -> move right
- 2 -> move down
- 3 -> move left

To execute an action, simply pass it as an argument to the env.step method. Information about the action space is stored in the env.action_space property which is of Discrete(4) class. This means that in this case it only consists of an element in the range [0,4), unlike the state space seen above.

```
print(f"An example of a valid action is: {env.action_space.sample()}")
print(f"The action state is of type: {env.action_space}")
```

Trajectories and episodes

An episode is a trajectory that goes from the initial state of the process to the final one:

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

where T is the terminal state.

Let's generate a whole episode in code:

```
env = Maze()
state = env.reset()
episode = []
done = False
while not done:
    action = env.action_space.sample()
    next_state, reward, done, extra_info = env.step(action)
    episode.append([state, action, reward, done, next_state])
    state = next_state
env.close()

print(f"Congrats! You just generated your first episode:\n{episode}")
```

Rewards and returns

The return associated with a moment in time t is the sum (discounted) of rewards that the agent obtains from that moment. We are going to calculate G_0, that is, the return to the beginning of the episode:

$$G_0 = R_1 + \gamma R_2 + \gamma^2 R_3 + \ldots + \gamma^{T-1} R_T$$

Let's assume that the discount factor $\gamma = 0.99$:

```
env = Maze()
state = env.reset()
done = False
gamma = 0.99
G_0 = 0
t = 0
while not done:
    action = env.action_space.sample()
    _, reward, done, _ = env.step(action)
    G_0 += gamma ** t * reward
    t += 1
env.close()

print(
    f"""It took us {t} moves to find the exit,
    and each reward r(s,a)=-1, so the return amounts to {G_0}""")
```

Policy

Policy

A policy is a function $\pi(a|s) \in [0, 1]$ that gives the probability of an action given the current state. The function takes the state and action as inputs and returns a float in [0,1].

Since in practice we will need to compute the probabilities of all actions, we will represent the policy as a function that takes the state as an argument and returns the probabilities associated with each of the actions. Thus, if the probabilities are:

[0.5, 0.3, 0.1]

we will understand that the action with index 0 has a 50% probability of being chosen, the one with index 1 has 30% and the one with index 2 has 10%.

Let's code a policy function that chooses actions randomly:

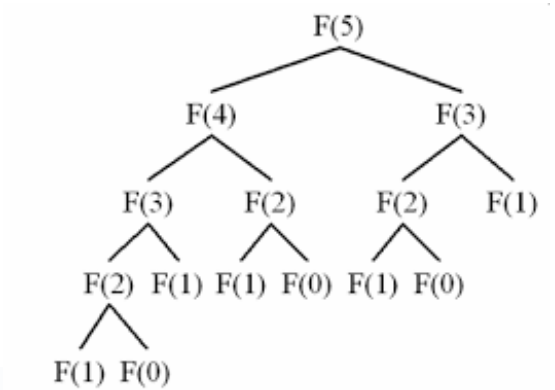
```
def random_policy(state):
    return np.array([0.25] * 4)
```

Dynamic Programming

Definition

- Methods that find the solution to a problem by breaking it down into smaller, easier problems.

- Remember Fibonacci sequence problem from your CS class



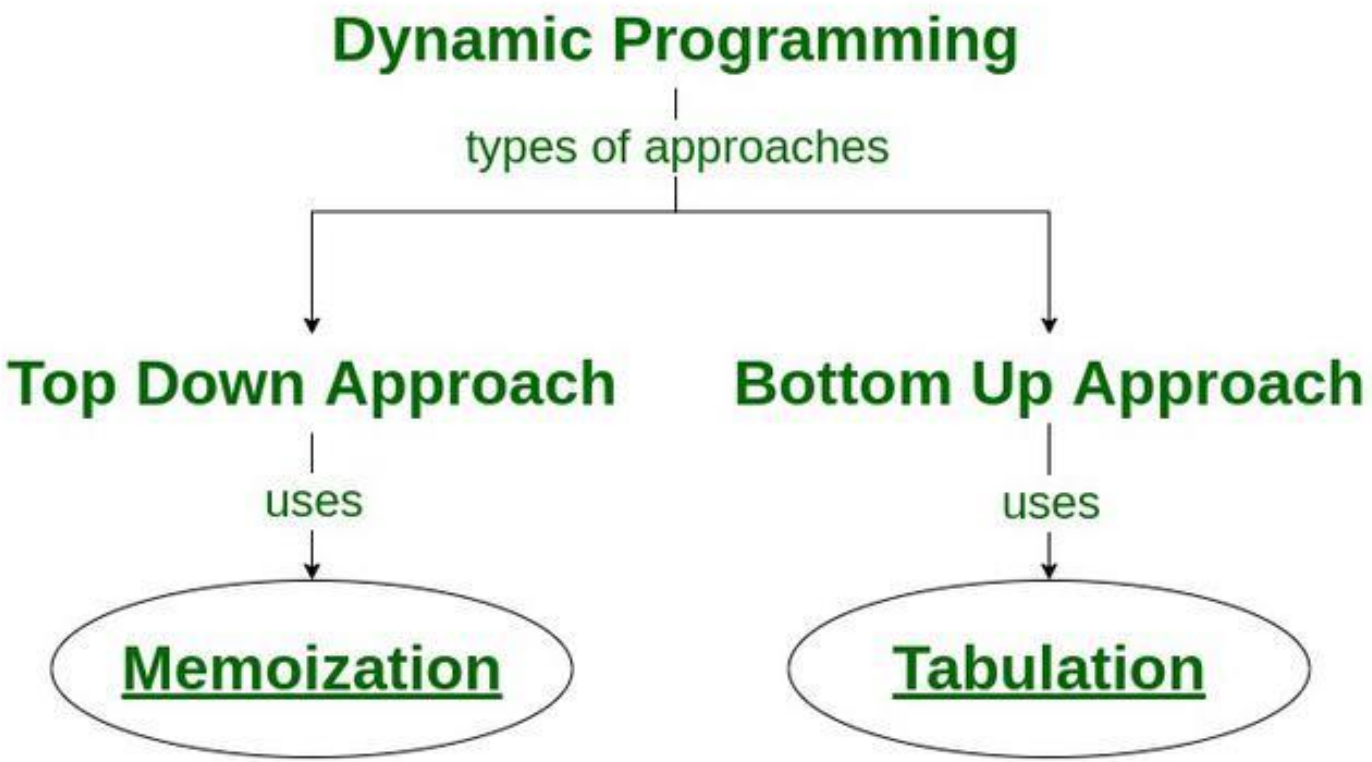

```
fn(5);
fn(4) + fn(3);
fn(3) + fn(2) + fn(2) + fn(1)
fn(2) + fn(1) + fn(1) + fn(0) + fn(1) + fn(0) + fn(1)
```

```
function fn (n) {
  if (n === 0) {
    return 1;
} else if (n === 1) {
    return 1;
} else {
    return fn(n-1) + fn(n-2);
}
```

Base case

The recursive relationship

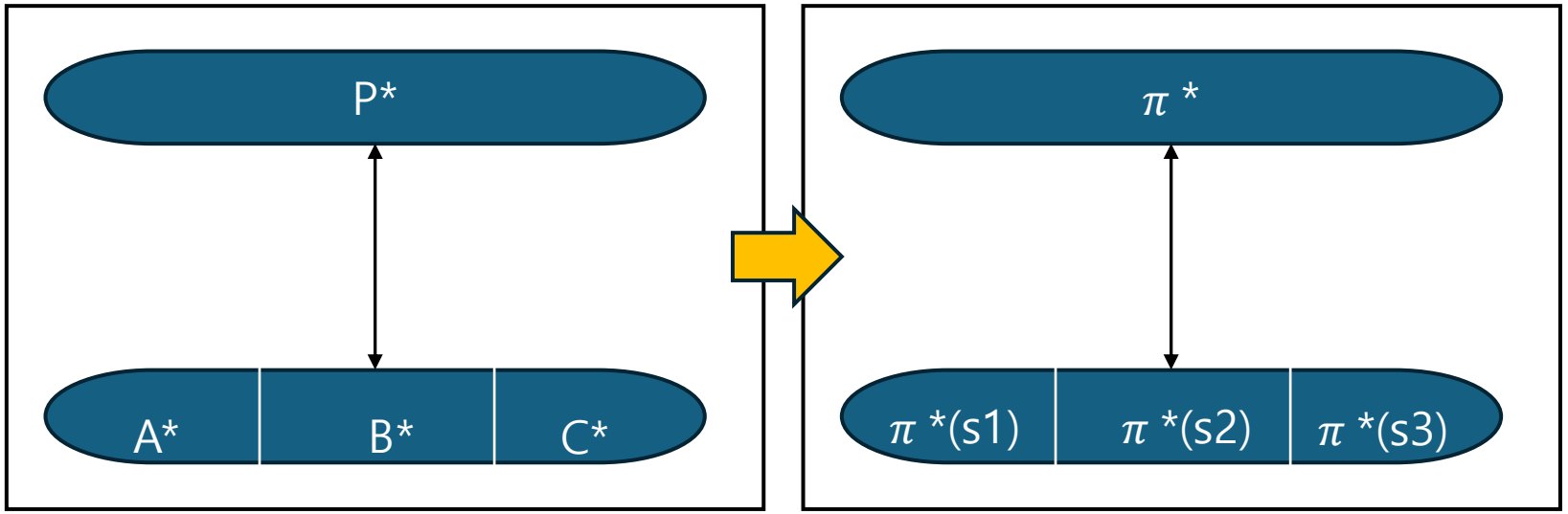
Memoization



		i \ j	0	1	2	3	4	5	6	7	8	9	10	11
		0	0	0	0	0	0	0	0	0	0	0	0	0
$\omega_1=1$	$V_1=1$	1	0	1	1	1	1	1	1	1	1	1	1	1
$\omega_2=2$	$V_2=6$	2	0	1	6	7	7	7	7	7	7	7	7	7
$\omega_3=5$	$V_3=18$	3	0	1	6	7	7	18	19	24	25	25	25	25
$\omega_4=6$	$V_4=22$	4	0	1	6	7	7	18	22	24	28	29	29	40
$\omega_5=7$	$V_5=28$	5	0	1	6	7	7	18	22	28	29	34	35	40

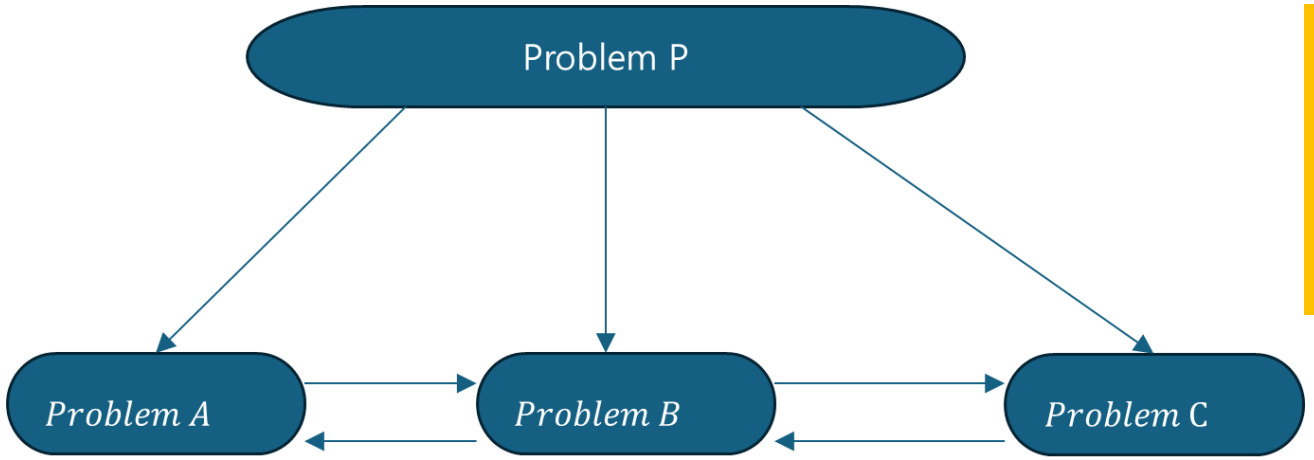
Optimal substructure

- In finding the solution to each of its subproblems and joining those individual solutions, we'll have found the optimal solution to the original problem



Overlapping subproblems

- The solution to the subproblems are mutually dependent
- The optimal solution to the problem A will be dependent on problem B and the problem B will be dependent on both problem A and problem C



The optimal solution to all subproblems produces the optimal solution to the original problem

Find v^*, q^*,

Our task: Find $\pi *$

- We can guide and structure the search for the policy using value functions
- The optimal policy takes actions based on state or q-values
- Therefore, to find the optimal policy, we need to find the optimal values

Our task: Find $\pi *$

- If we find the optimal value for each state independently, then we'll have the optimal value function for the overall problem

$$\pi * \leftrightarrow \pi * (s, a)$$
$$q * \leftrightarrow q * (s, a)$$
$$v * \leftrightarrow v * (s)$$

State value function(v)

- State value, following policy π :

$$V_{\pi}(s) = E[G_t | S_t = s]$$

$$V_{\pi}(s) = E[R_{t+1} + \gamma * R_{t+2} + \gamma^2 * R_{t+3} + \ldots + \gamma^{T-t-1} R_T \,|\, St = s]$$

State-action value function(q)

- State-action value, following policy π :

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

$$Q_{\pi}(s,a) = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-t-1} R_T \mid S_t = s, A_t = a]$$

Bellman Equation for v(s)

- Bellman Eq. to search for optimal policy to solve a control task

$$
\begin{aligned}
V_\pi(s) &= E[G_t | S_t = s] \\
&= E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-t-1} R_T \,|\, S_t = s] \\
&= E[R_{t+1} + \gamma G_{t+1} \,|\, S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V_\pi(s')]
\end{aligned}
$$

Expected return is, the prob. of taking each action following the policy, multiplied by the return we expect to get from taking that action.

The return is, the prob. of reaching each possible successor state, multiplied by the reward obtained upon reaching that state, plus the discounted value of that state

Notice we discovered a recursive relationship btw. The value of one state and the values of other states

Bellman Equation for q(s, a)

Bellman Eq. to search for optimal policy to solve a control task

$$
\begin{aligned}
Q_{\pi}(s,a) &= E[\,G_t \,|S_t = s, A_t = a] \\
&= E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-t-1} R_T \,|\, S_t = s, A_t = a] \\
&= E[R_{t+1} + \gamma G_{t+1} \,|\, S_t = s, At = a] \\
&= \sum_{s',r} p(s', r|s, a)\ [\, r + \gamma \sum_{a'} \pi(a'|s')\, Q_{\pi}(s', a')\,]
\end{aligned}
$$

Expected return is the prob. of each successor state, knowing that we have chosen action a.

Multiplied by ①the reward obtained upon reaching that successor state, ② plus the discounted sum of q values of each action in the successor state, ③ weighted the prob. of choosing that action by the policy

Recursive relationship is expressed as Q value in terms of other Q values

Solving a MDP

- The value of a state is precisely the expected return
- Solving a control task consists of maximizing the expected return
- Solving a task involves maximizing the value of every state
- The optimal value of a state is the expected return following the optimal policy:

$$\mathrm{v}*(s) = E[G_t | S_t = s]$$

$$\mathrm{q}^*(s,a) = E[G_t | S_t = s, A_t = a]$$

Solving a MDP

- To maximize those return, we must find the optimal policy.
  (The policy that takes optimal action in all state)

- The optimal π^* policy is precisely the one that chooses actions that maximize v(s) or q(s, a), the expected return:

$$\pi^*(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v*(s')]$$

$$\pi^*(s) = \arg\max_a q*(s, a)$$

Solving a MDP

- Dilemma is ,

- To find the optimal policy $\pi*$,
  we must know the optimal values

- To find the optimal v∗ or q∗ values,
  we must know the optimal policy

Bellman Optimality Equations

- The optimal policy will always choose the action that maximizes the expected return

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

The prob. of reaching each successor state by taking the optimal action,

multiplied by the reward achieved by reaching that state, plus the discounted optimal value of that state

following the optimal policy π^*

Bellman Optimality Equations

- The optimal policy will always choose the action that maximizes the expected return

$$q*(s,a) = \sum_{s',r} p(s',r|s,a)\ [r + \gamma \max_{a'} q*(s',a')]$$

Optimal Q value for an action in a state is the weighted sum of returns

obtained by reaching each of possible successor state, weighted by the prob. of reaching that successor state.

The return is reward achieved upon reaching the successor state, plus the maximum Q value among the actions for that achieved state

following the optimal policy $\pi*$

Find π^*

Update Rules

- DP turns Bellman equations into update rules

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

$$V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V_*(s')]$$

Update Rules

- Sweep the state space and update the estimated value of each state according to:

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

- Each time we update the estimated value of a state, we'll have better estimates for the related values, therefore, the new estimate will be more accurate than the old one

Value Iteration

Value Iteration

- We want to find the optimal policy $\pi^*(s)$:

$$\pi^*(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v*(s')]$$

...but for that we have to find v*

Value Iteration

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

- Initial estimate doesn't have to be good
- We'll keep a table with the estimated values of each state
- We'll go state by state improving these estimates according to the rule
- Repeat this process as many times as necessary until we estimates are very close to the values

Algorithm 2 Value Iteration

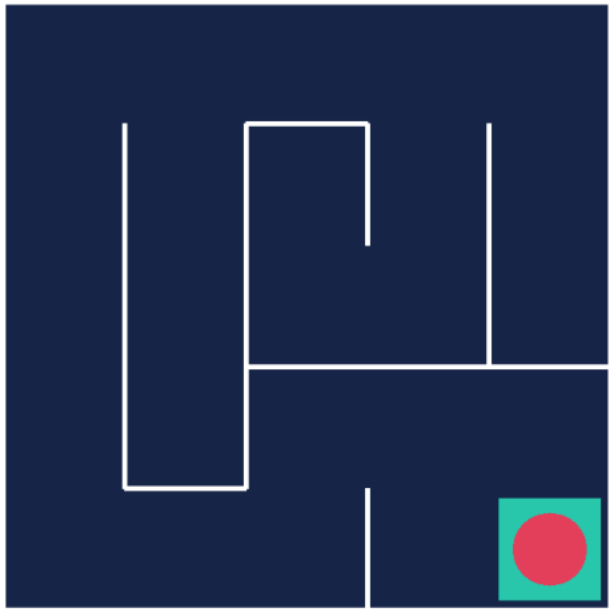
Input: $\theta > 0$ tolerance parameter, γ discount factor
Initialize $V(s)$ arbitrarily, with $V(terminal) = 0$
repeat
    $\Delta \leftarrow 0$
    for $s \in S$ **do**
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_{a \in A(s)} \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end for
until $\Delta > \theta$
Output: π: greedy policy w.r.t. $V(s)$

Value Iteration

t = 0

- Every move, the agent gets -1. The goal is to find the shortest path.
- This env. Is deterministic

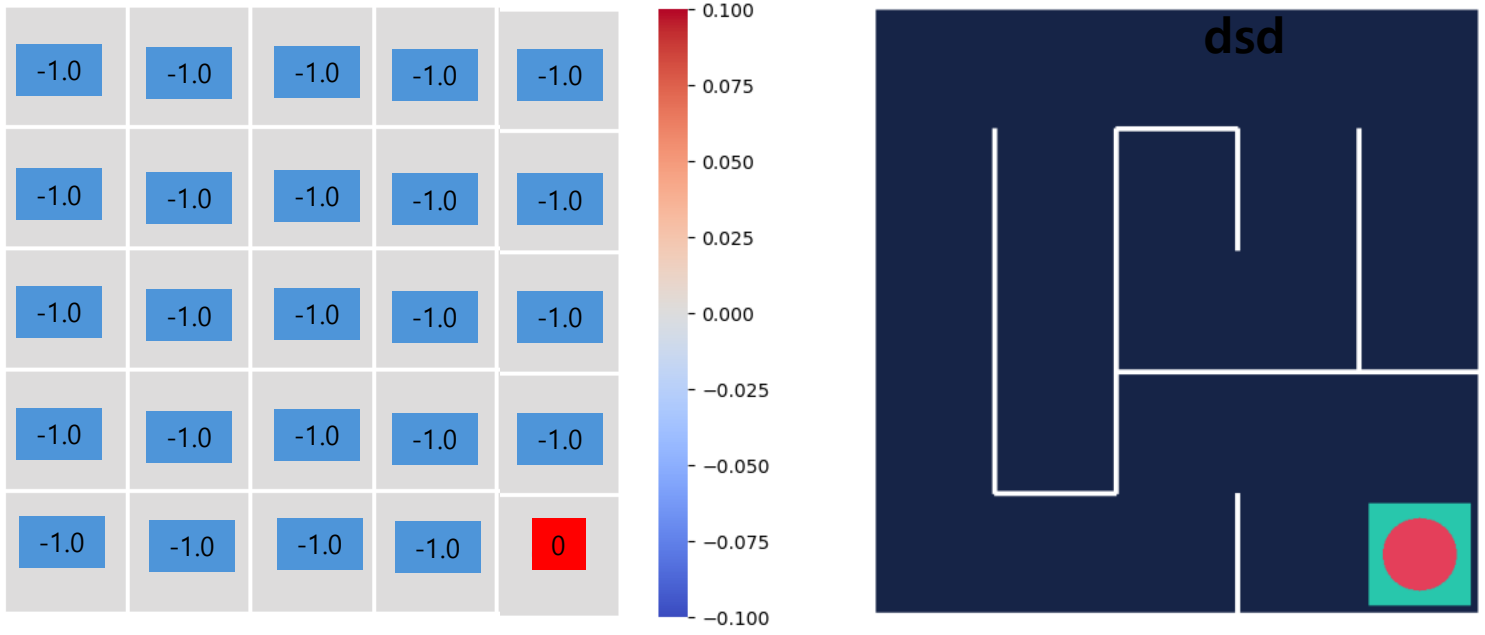
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00



$$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$$

Value Iteration

t = 1

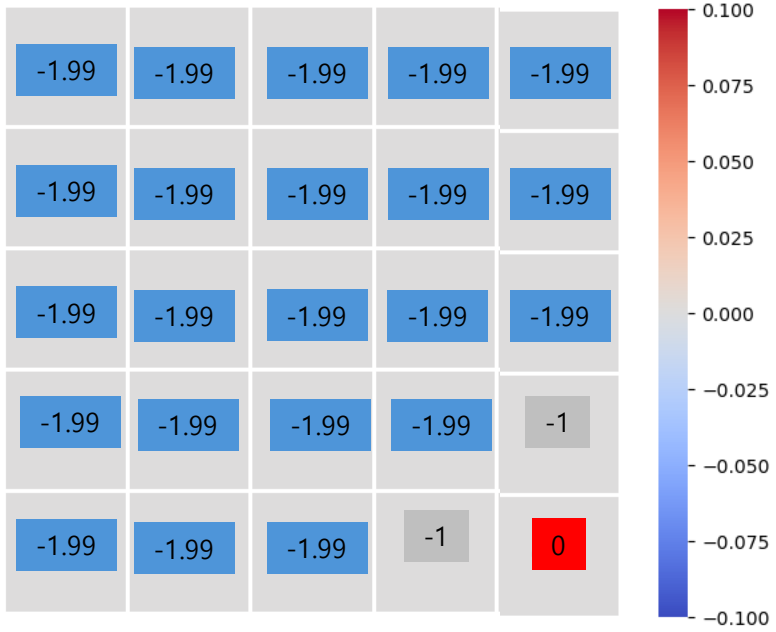


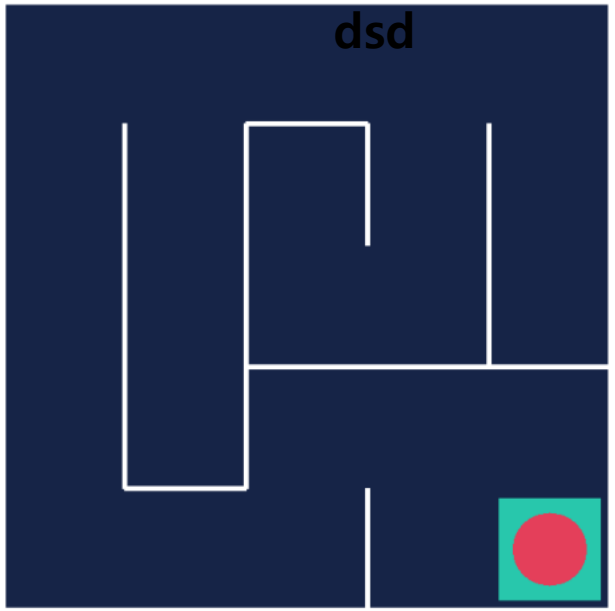
$$V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$$

Value Iteration

t = 2

Every move, the agent gets -1
The goal is to find the shortest path
This env. Is deterministic

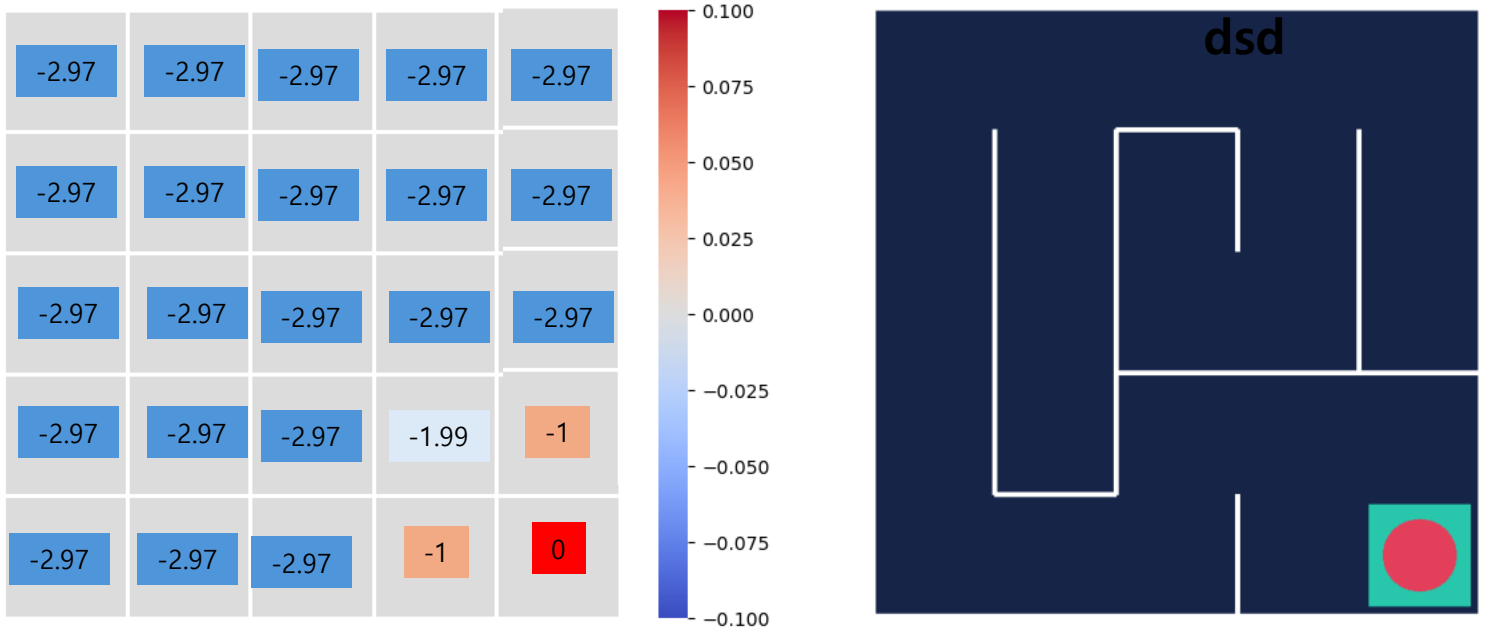




$$V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$$

Value Iteration

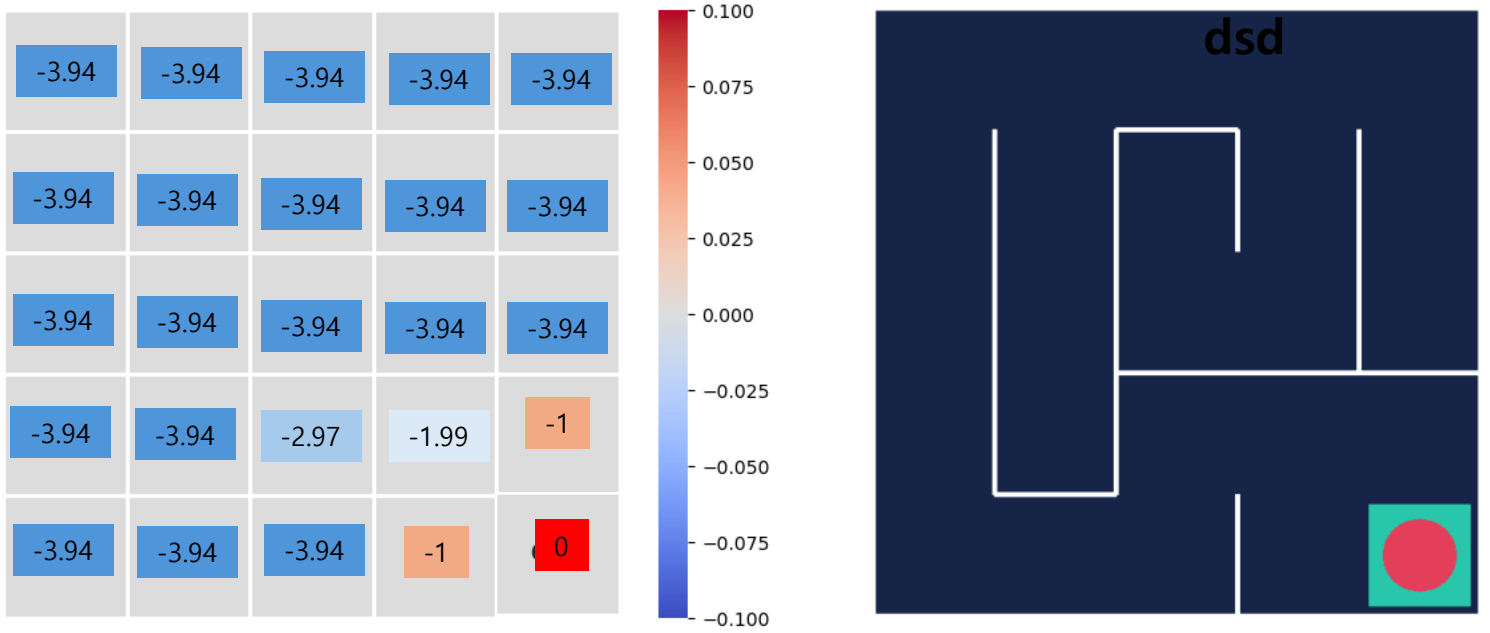
t = 3



$$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$$

Value Iteration

t = 4



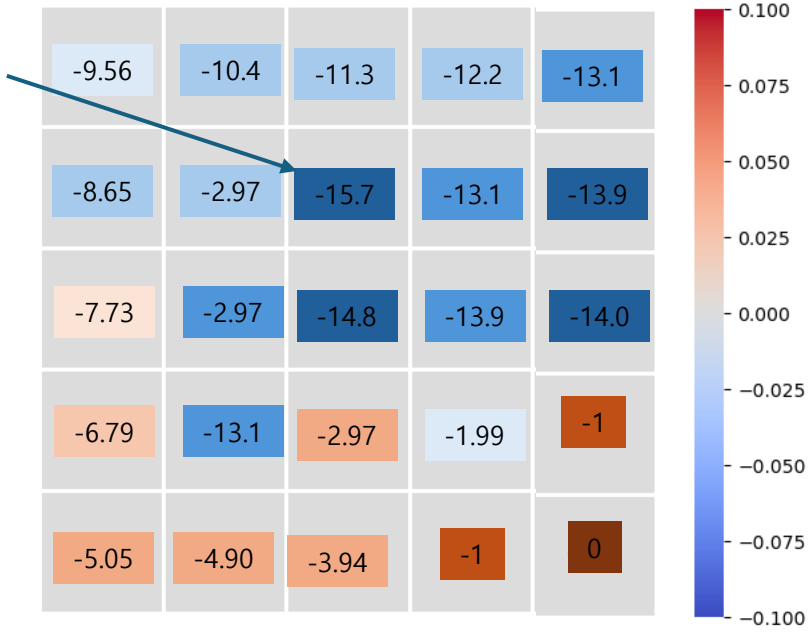
$$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$$

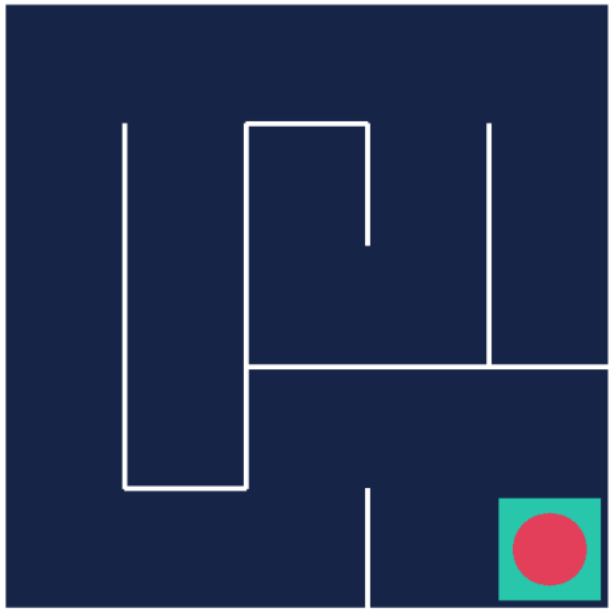
Value Iteration

t = 18

- Found the optimal value for all state.
- Note the farther a state is from the goal, the lower its value, because the agent collects negative rewards every time it moves

The lowest one because its farthest (17 steps to reach it)





$$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$$

Value Iteration

$$V(s) \leftarrow \max_{a} \sum_{s',r} p(s', r|s, a)[r + \gamma V*(s')]$$

- Initial estimate doesn't have to be good
- We'll keep a table with the estimated values of each state
- We'll go state by state improving these estimates according to the rule
- Repeat this process as many times as necessary until we estimates are very close to the values

Code Exercise

Import the necessary software libraries:

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

from envs import Maze
from utils import plot_policy, plot_values, test_agent
```

Initialize the environment

```
env = Maze()
```

```
frame = env.render(mode='rgb_array')
plt.figure(figsize=(4,4))
plt.axis('off')
plt.imshow(frame)
```

```
print(f"Observation space shape: {env.observation_space.nvec}")
print(f"Number of actions: {env.action_space.n}")
```

```
Observation space shape: [5 5]
Number of actions: 4
```

Code Exercise

Define the policy $\pi(\cdot|s)$

Create the policy $\pi(\cdot|s)$

```
policy_probs = np.full((5, 5, 4), 0.25)
```

```
def policy(state):
    return policy_probs[state]
```

Test the policy with state (0, 0)

```
action_probabilities = policy((0,0))
for action, prob in zip(range(4), action_probabilities):
    print(f"Probability of taking action {action}: {prob}")
```

```
Probability of taking action 0: 0.25
Probability of taking action 1: 0.25
Probability of taking action 2: 0.25
Probability of taking action 3: 0.25
```

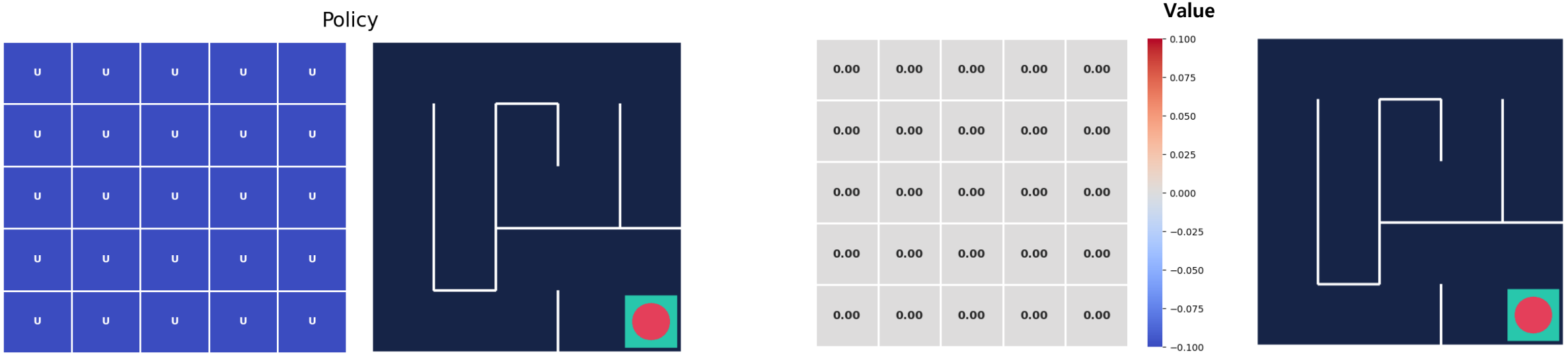
See how the random policy does in the maze

```
test_agent(env, policy, episodes=1)
```

See how the random policy plays in the env.

```
plot_policy(policy_probs, frame)
```

Code Exercise



Code Exercise

```python
def value_iteration(policy_probs, state_values, theta=1e-6, gamma=0.99):
    delta = float('inf')

    while delta > theta:
        delta = 0
        for row in range(5):
            for col in range(5):
                old_value = state_values[(row, col)]
                action_probs = None
                max_qsa = float('-inf')

                for action in range(4):
                    next_state, reward, _, _ = env.simulate_step((row, col), action)
                    qsa = reward + gamma * state_values[next_state]
                    if qsa > max_qsa:
                        max_qsa = qsa
                        action_probs = np.zeros(4)
                        action_probs[action] = 1.

                state_values[(row, col)] = max_qsa
                policy_probs[(row, col)] = action_probs

                delta = max(delta, abs(max_qsa - old_value))
```

Without interacting with env. Get the information about what would happen if we did

Deterministic policy (assign 100% prob. to the optimal action)

Update Policy

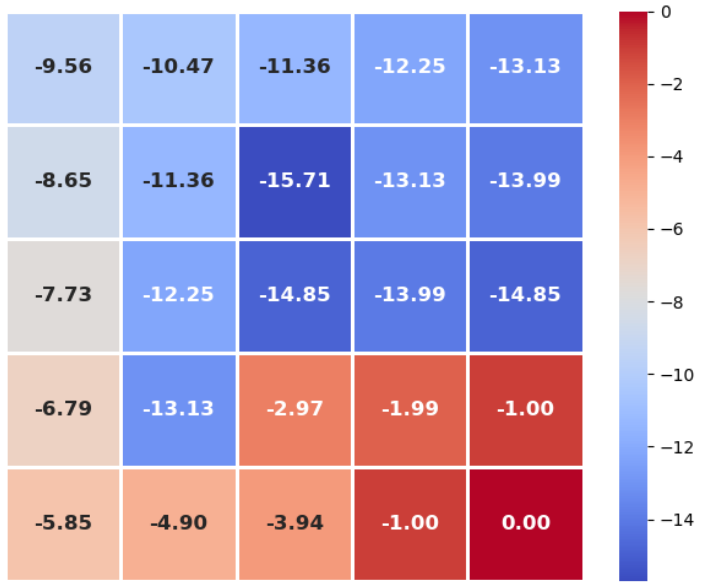
```python
value_iteration(policy_probs, state_values)
```

Code Exercise

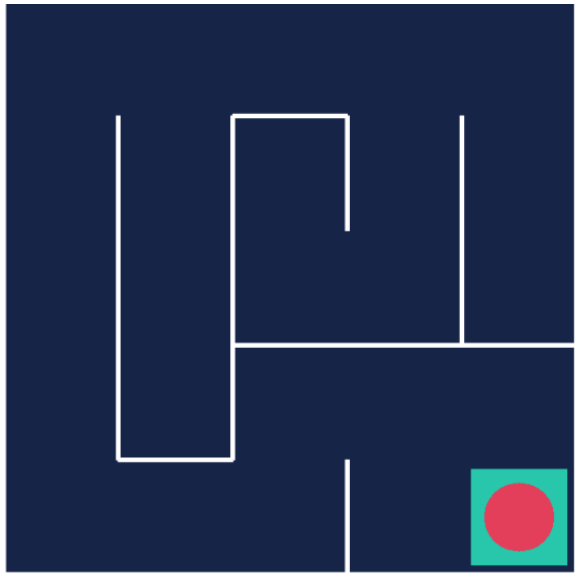
$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$

π: greedy policy w.r.t. $V(s)$

-9.56	-10.47	-11.36	-12.25	-13.13
-8.65	-11.36	-15.71	-13.13	-13.99
-7.73	-12.25	-14.85	-13.99	-14.85
-6.79	-13.13	-2.97	-1.99	-1.00
-5.85	-4.90	-3.94	-1.00	0.00



Value table



D	L	L	L	L
D	U	D	U	U
D	U	R	U	U
D	U	R	R	D
R	R	U	R	R

Policy table

Optimal policy for all states

Setup Code Env.

- Git clone https://github.com/parkjin-nim/rl_lecture.git
- conda create -n <your env. name> -f environment.yml
- Open Jupyter notebook
- Open 2.DP/value_iteration.ipynb