3강.
Dynamic Programming
(Part 2)

Contents

- Dynamic Programming Review
- Policy Iteration
- Code Exercise
- Dynamic Programming Limitations
- Generalized Policy Iteration

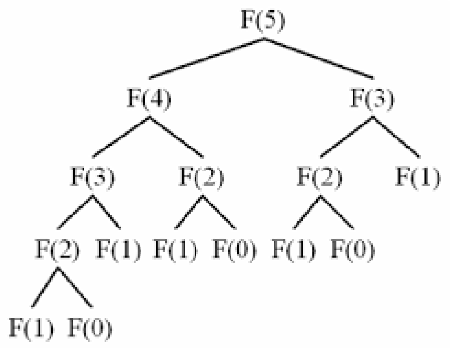
Dynamic Programing Review

Definition

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

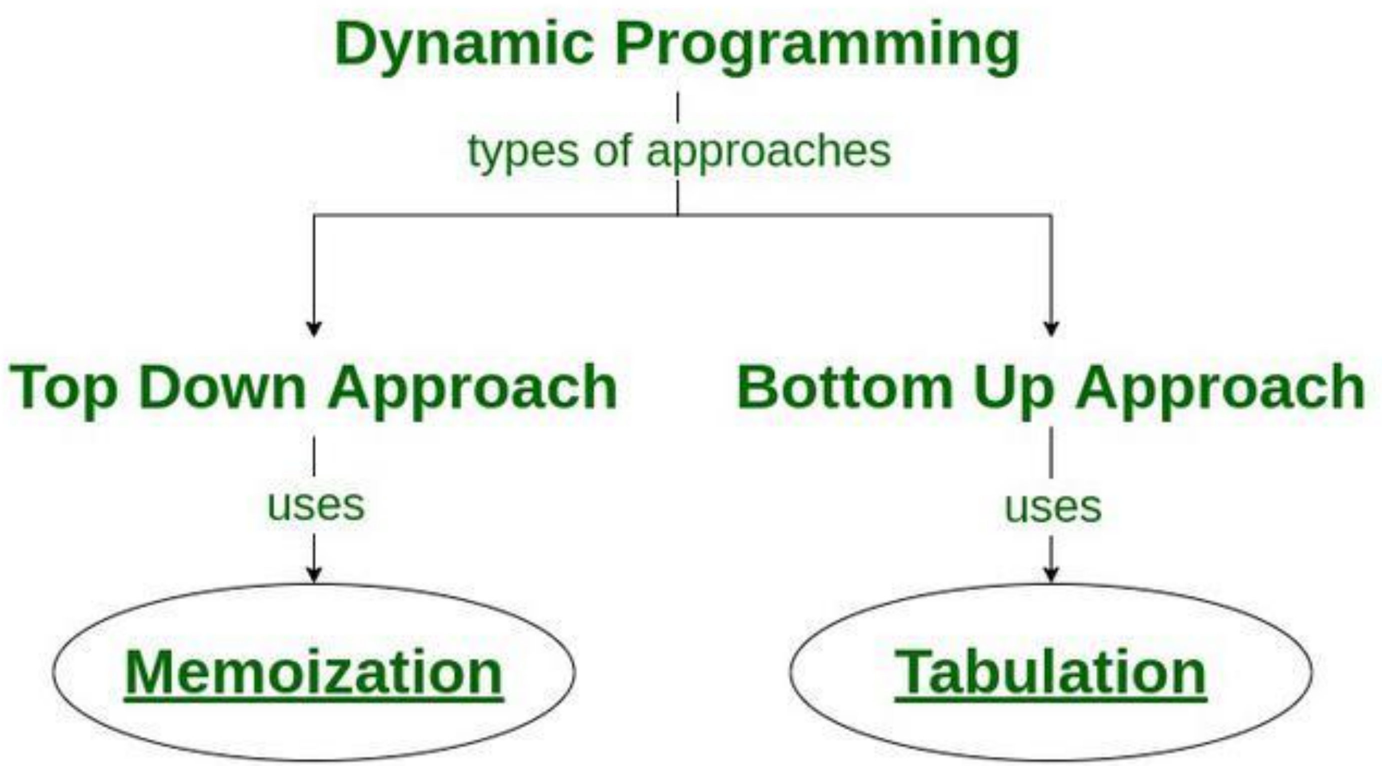
- Remember Fibonacci sequence problem from your CS class

```
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);
}
```

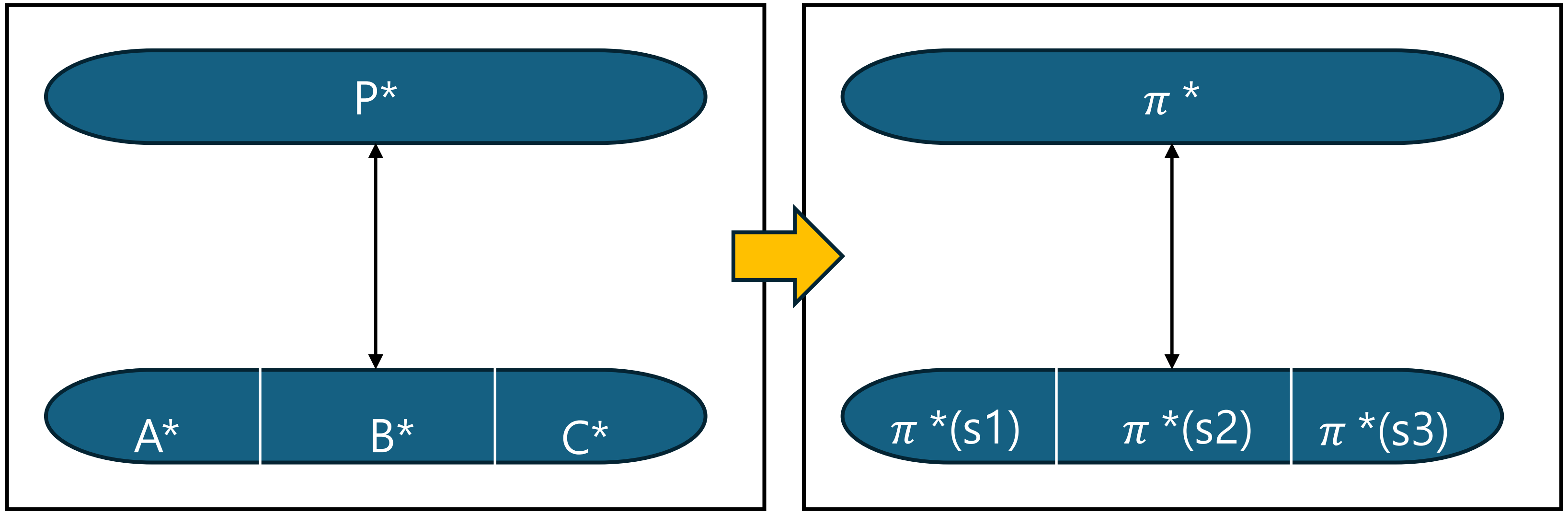
Memoization



		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

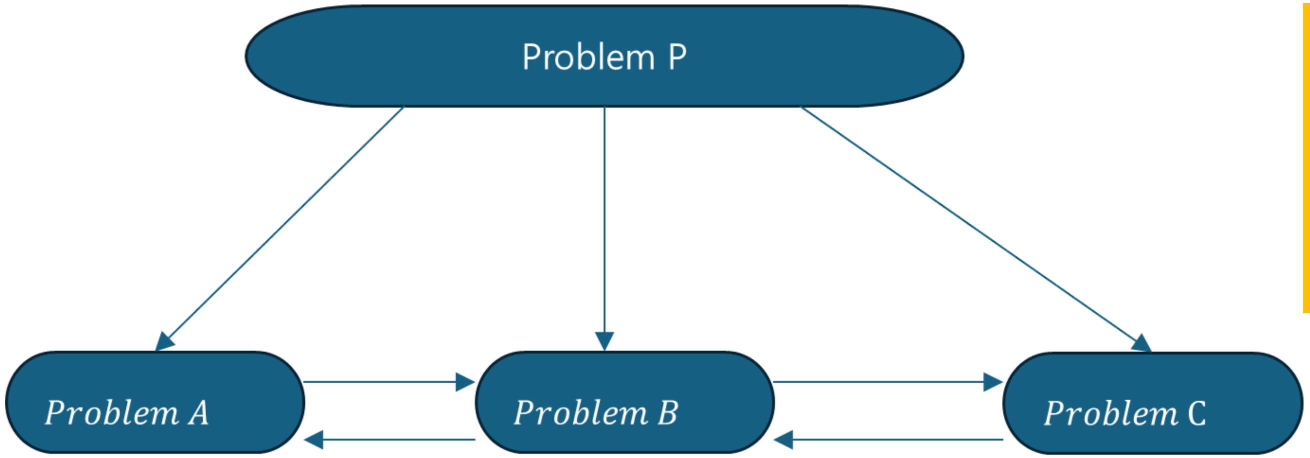
Optimal substructure

- 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

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

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

Find v^*, q^*,

State value(v) vs state-action(q) value

- 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, following policy π :

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

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

Bellman Equation for v(s)

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

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

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

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

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:

$$v*(s) = E[G_t | S_t = s]$$

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

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:

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

$$\pi^*(\mathrm{s}) = \underset{a}{\arg\max}\; \mathrm{q}{*}(s, a)$$

Solving a MDP

- Dilemma is ,

- $\begin{cases} \text{To find the optimal policy } \pi* \text{ ,} \\ \text{we must know the optimal values} \\ \\ \text{To find the optimal v* or q* values,} \\ \text{we must know the optimal policy} \end{cases}$

Bellman Optimality Equations

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

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

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 π^*

Update Rules

- DP turns Bellman equations into update rules

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

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

Update Rules

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

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

- 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

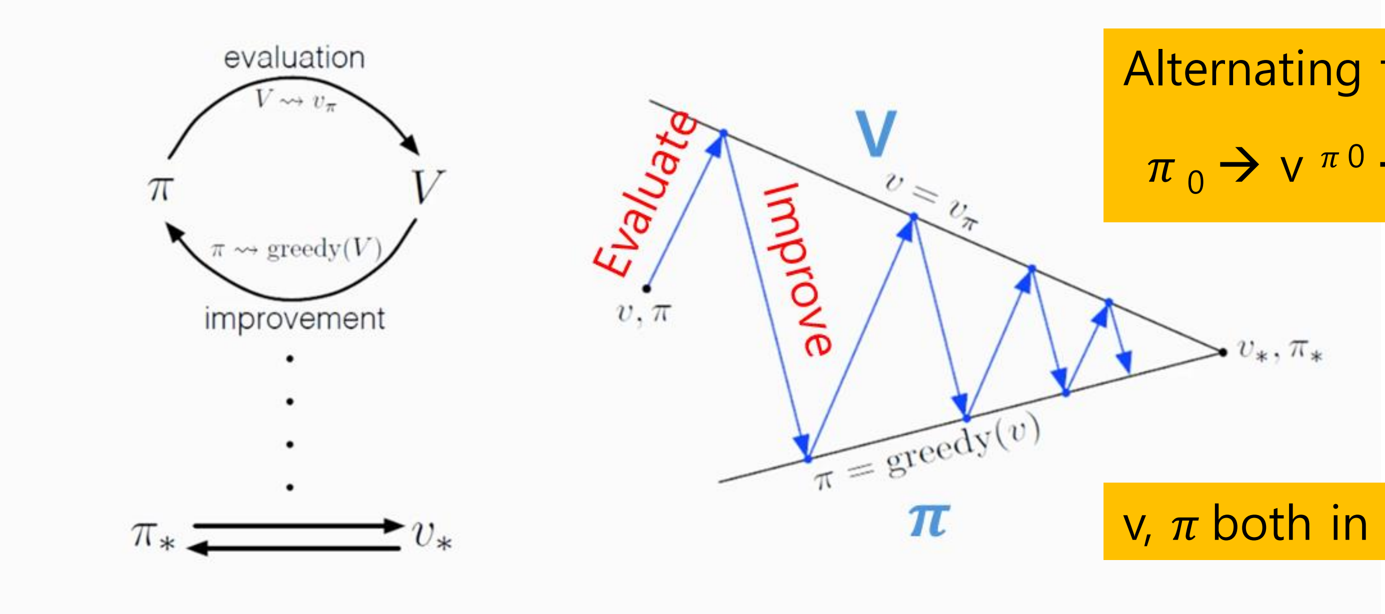
$$V(s) \leftarrow \max_{a} \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$$

- 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

Policy Iteration

Policy Iteration

- Policy Iteration solves control tasks, but besides that, it will serve as inspiration to design the vast majority of reinforcement learning
- A process that alternately improves the estimated values and the policy



Alternating two processes compete each other

$\pi_0 \rightarrow v^{\pi 0} \rightarrow \pi_1 \rightarrow v^{\pi 1} \rightarrow \pi_2 \rightarrow v^{\pi 2} \ldots \rightarrow \pi_n$

v, π both in competition and collaboration

Policy Iteration

Algorithm 2 Policy Iteration

Input: $\theta > 0$ tolerance parameter, γ discount factor
Initialize $V(s)$ and $\pi(a|s)$ arbitrarily
while policy-stable $= false$ **do**

    Policy Evaluation:
    while $\Delta > \theta$ **do**
        $\Delta \leftarrow 0$
        for $s \in S$ **do**
            $v \leftarrow V(s)$
            $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
        end for
    end while

    Policy Improvement:
    policy-stable $= true$
    for $s \in S$ **do**
        old-action $\leftarrow \pi(s)$
        $\pi(s) \leftarrow \arg\max_{a \in A(s)} \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
        if old-action $\neq \pi(s)$ **then**
            policy-stable $\leftarrow false$
        end if
    end for

end while
Output: Optimal policy $\pi(a|s)$ and state values $V(s)$

Iterative policy evaluation

- The Bellman equation:

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

- The update rule:

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

- Iteratively approximates the values of a given policy, v_π
- Each iteration gets closer to $\mathbf{V}_{\boldsymbol{\pi}}$

$$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_{\boldsymbol{\pi}}$$

Iterative policy evaluation

- Each iteration gets closer to $\mathbf{V}_{\boldsymbol{\pi}}$

$$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_{\boldsymbol{\pi}}$$

while $\Delta > \theta$ **do**
    $\Delta \leftarrow 0$
    for $s \in S$ **do**
        $v \leftarrow V(s)$
        $V(s) \leftarrow \boxed{\sum_a \pi(a|s)} \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end for
end while

Update the state values according to probabilities that the policy assigns to each action

Iterative policy improvement

- We want to find the optimal policy π*(s) :

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

- The update rule:

$$\pi(\mathrm{s}) \leftarrow \arg\max_{a} \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$$

- Iteratively improve π

Iterative policy improvement

- Does the policy improve if we change the first action?

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

- π and π' differ only in the action a they take in state s.

$$\text{If } q_\pi(s, \pi'(s)) \geq v_\pi(s), \text{ then } v_{\pi'}(s) \geq v_\pi(s)$$

Iterative policy improvement

- Each iteration gets closer to π^*

$$\pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \dots \rightarrow \pi_{\boldsymbol{n}}$$

policy-stable $= true$
for $s \in S$ **do**
  old-action $\leftarrow \pi(s)$
  $\pi(s) \leftarrow \arg\max_{a \in A(s)} \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
  if old-action $\neq \pi(s)$ **then**
    policy-stable $\leftarrow false$
  end if
end for

Compare PI vs. VI

1. Initialization
   $v(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\quad \Delta \leftarrow 0$
   $\quad$ For each $s \in \mathcal{S}$:
   $\quad\quad temp \leftarrow v(s)$
   $\quad\quad v(s) \leftarrow \sum_{s'} p(s'|s,\pi(s))\left[r(s,\pi(s),s') + \gamma v(s')\right]$
   $\quad\quad \Delta \leftarrow \max(\Delta, |temp - v(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $\quad temp \leftarrow \pi(s)$
   $\quad \pi(s) \leftarrow \arg\max_a \sum_{s'} p(s'|s,a)\left[r(s,a,s') + \gamma v(s')\right]$
   $\quad$ If $temp \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return v and π; else go to 2

finding optimal value function

Initialize array v arbitrarily (e.g., $v(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
$\quad \Delta \leftarrow 0$
$\quad$ For each $s \in \mathcal{S}$:
$\quad\quad temp \leftarrow v(s)$
$\quad\quad v(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v(s')]$
$\quad\quad \Delta \leftarrow \max(\Delta, |temp - v(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
$\boxed{\pi(s) = \arg\max_a \sum_{s'} p(s'|s,a)\left[r(s,a,s') + \gamma v(s')\right]}$

Figure 4.5: Value iteration.

one policy update (extract policy from the optimal value function

<u>Update the state values</u> using the action that maximize the return

Policy-iteration is more computationally efficient as it often takes considerably fewer number of iterations to converge although each iteration is more computationally expensive.

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)
```

Code Exercise

```python
def policy_evaluation(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)]
                new_value = 0
                action_probabilities = policy_probs[(row, col)]

                for action, prob in enumerate(action_probabilities):
                    next_state, reward, _, _ = env.simulate_step((row, col), action)
                    new_value += prob * (reward + gamma * state_values[next_state])

                state_values[(row, col)] = new_value

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

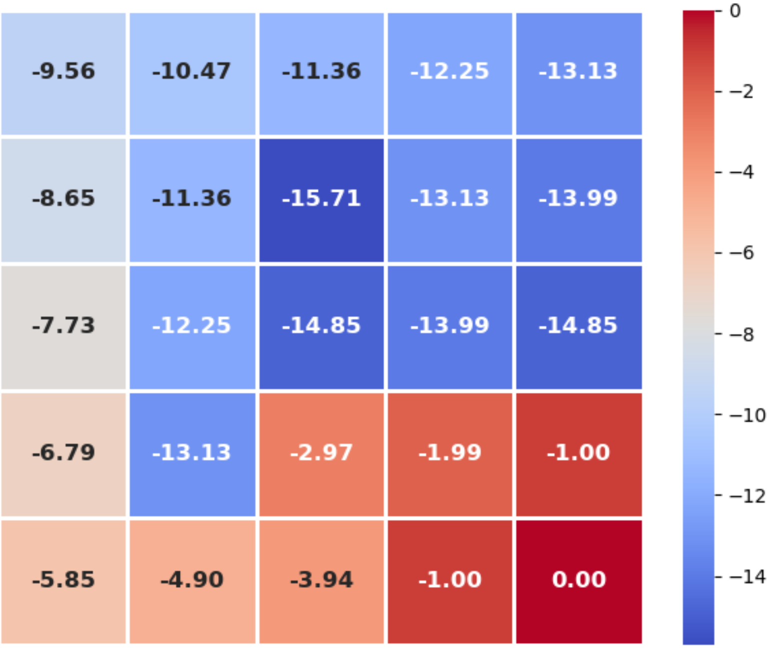
Code Exercise

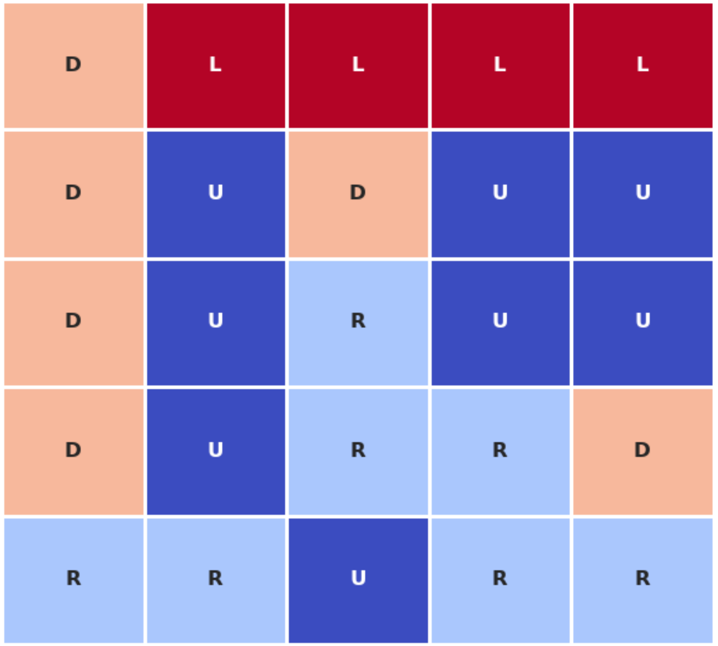
```python
def policy_improvement(policy_probs, state_values, gamma=0.99):

    policy_stable = True
    for row in range(5):
        for col in range(5):
            old_action = policy_probs[(row, col)].argmax()

            new_action = 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
                    new_action = action

            action_probs = np.zeros(4)
            action_probs[new_action] = 1.
            policy_probs[(row, col)] = action_probs

            if new_action != old_action:
                policy_stable = False

    return policy_stable
```

Policy Iteration

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

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





Value table vs. Policy table (Same as VI)

Dynamic Programming Limitations

Dynamic Programming Limitations

- We need to know in advance how the state changes and what rewards we get from performing each action in each state

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

- Assumes the knowledge of environment
- One big limitation is that it needs a perfect model of the environment

Dynamic Programming Limitations

- Needs to access to the state transition dynamics of environment
- Takes into an account every possible outcome of taking an action and uses it to update the estimated values

- Needs to know the result of taking every action in every state in advance without having to perform that action
- Note that Dynamic Programing solves problems using expected values, not trial and error

Dynamic Programming Limitations

- Dynamic Programing has high computational cost for control:
  - To solve the task, we have to update over and over again

- In each sweep we update all the states
  - Complexity grows very rapidly with the number of states
  - But, real life control problems have a vast or even infinite number of states

Dynamic Programming Limitations

- In most tasks, we won't have a perfect model of the environment with all state transitions

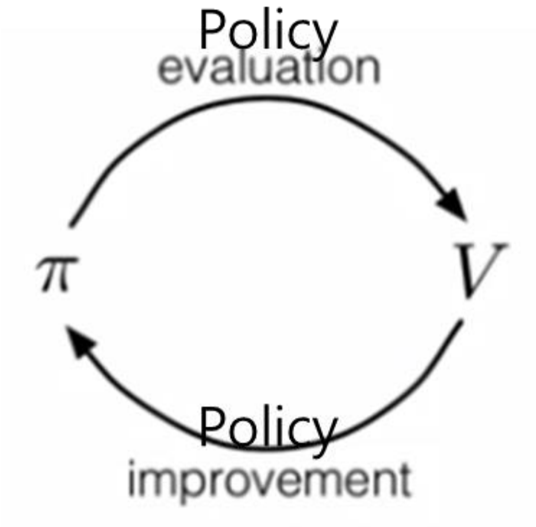
- Most control tasks have many factors affecting their dynamics (some of them random).

- Then, why we learn Dynamic Programing?

  Thanks to it we have a strategy to design better algorithm.

Generalized Policy Iteration

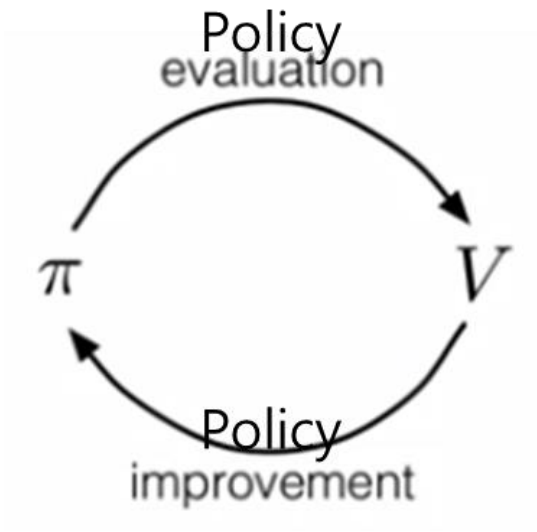
- Policy iteration results in the following iterative process:



- From the next class, algorithms we learn won't have a model. They interact with env. trying to solve the task by trial and error.
- However, they still follow the two alternating processes

Generalized Policy Iteration

- Next, Model-free RLs works without the model.



The difference is that the value updates will be made using experience collected from the environment

- Model-free methods use experience samples collected by the agent interacting with the environment (by trial and error) to update the estimated value

Generalized Policy Iteration

- They try to replicate the results of dynamic programming, but in a more efficient way and without the need of a model of the environment dynamics.

- The process serves as a template followed by the rest of the RL methods