Midterm

411086036 電機四 鄧曉晴

1.1. Run and debug the provided code. You must clearly identify and highlight any corrections or modifications you make during the debugging process.

(1) 第 90 行

transition_probabilities，應為逐步累積加大，改為：

transition_probabilities[(state, action, new_discrete_state)] += 1 / NUMBER_OF_SAMPLES。

```
76      def compute_transition_probabilities(self):
77          transition_probabilities = defaultdict(lambda: 0)
78          for state in self.states:
79              discrete_position, discrete_velocity = state
80              for action in self.actions:
81                  for _ in range(NUMBER_OF_SAMPLES):
82                      continuous_position = sample_position_from_discretized(discrete_position, self.discretization_position)
83                      continuous_velocity = sample_velocity_from_discretized(discrete_velocity, self.discretization_velocity)
84
85                      new_velocity = compute_new_velocity(continuous_position, continuous_velocity, action)
86                      new_position = compute_new_position(continuous_position, new_velocity)
87
88                      new_discrete_state = discretize(new_position, new_velocity, self.discretization)
89
90                      #transition_probabilities[(state, action, new_discrete_state)] -= 1 / NUMBER_OF_SAMPLES
91                      transition_probabilities[(state, action, new_discrete_state)] += 1 / NUMBER_OF_SAMPLES
92          return transition_probabilities
```

(2) 102 行

Reward 邏輯還沒建立完善，我新增目標狀態：當位置≥0.5 時分配獎勵=0。

```
112     def compute_rewards(self):
113         rewards = defaultdict(lambda: -1)
114         for state in self.states:
115             for action in self.actions:
116                 for new_state in self.states:
117                     position = sample_position_from_discretized(new_state[0], self.discretization_position)
118                     if position >= 0.5:
119                         rewards[(state, new_state)] = 0
120
121         return rewards
```

(3) 188 行 action 改 state

第 191 行 for...in 語法錯誤

```
185 def get_policy(mdp, V):
186     policy = {}
187     for state in mdp.states:
188         policy[state] = max(mdp.actions, key=lambda action:
189             sum(mdp.transition_probabilities.get((state, action, new_state), 0) *
190                 (mdp.rewards.get((state, new_state), -1) + mdp.gamma * V[new_state])
191                 # in new_state for mdp.states))
192                 for new_state in mdp.states))
193     return policy
```

(4) 語法錯誤

多區塊修正為 for … in … 語法

第 52 行

```python
47 def map_states_to_continuous(states, discretization):
48     return [(
49                 sample_position_from_discretized(state[0], discretization[0]),
50                 sample_velocity_from_discretized(state[1], discretization[1])
51             )
52             # in state for states
53             for state in states]
```

第 65 行

```python
59 class MDP:
60     def __init__(self, discretization=(DISCRETIZATION_POSITION, DISCRETIZATION_VELOCITY)):
61         self.gamma = 0.99
62         self.discretization = discretization
63         self.discretization_position = discretization[0]
64         self.discretization_velocity = discretization[1]
65         #self.states = {(i, j) in i for range(self.discretization_position) in i for range(self.discretization_velocity)}
66         self.states = {(i, j) for i in range(self.discretization_position) for j in range(self.discretization_velocity)}
```

第 178、181 行

第 176,177 行新增.get()，得以成功傳遞參數到 function 中。

```python
171 def value_iteration(mdp, num_iterations=NUM_ITERATIONS):
172     V = {state: 0 for state in mdp.states}
173     for _ in tqdm.tqdm(range(num_iterations)):
174         for state in mdp.states:
175             V[state] = max(
176                 sum(mdp.transition_probabilities.get((state, action, new_state), 0) *
177                     (mdp.rewards.get((state, new_state), -1) + mdp.gamma * V[new_state])
178                     # in new_state for mdp.states
179                     for new_state in mdp.states)
180                 # in action for mdp.actions
181                 for action in mdp.actions
182             )
183     return V
```

1.2. Add detailed comments to the following components: compute_rewards, compute_transition_probabilities, value_iteration, get_policy, and the MDP class. Higher marks will be awarded for comments that effectively relate the code to the mathematical concepts discussed in Lessons 04 and 05.

對應講義對 MDP 之定義：

A Markov Decision Process is a tuple <S, A, P, R, γ>

- S is a finite set of states
- A is a finite set of actions
- P is a state transition probability matrix,
  $$\mathcal{P}_{ss'}^{a} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$$
- R is a reward function,
  $$\mathcal{R}_{s}^{a} = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$$
- γ is a discount factor $\gamma \in [0, 1]$

翻譯放在程式中：

```python
class MDP:
    def __init__(self, discretization=(DISCRETIZATION_POSITION, DISCRETIZATION_VELOCITY)):
        """
        在MDP中, 5 tuple (S, A, P, R, γ):
        - S: 狀態空間(離散化的位置和速度)。
        - A: 動作空間(0: 減速、1: 無動作、2: 加速)。
        - P: 轉移概率 P(s'|s,a), func.計算。
        - R: 獎勵函數 R(s,s'), func.計算。
        - γ: 折扣因子,平衡即時與未來獎勵。

        參數:
            discretization: 狀態空間, 所有離散化的 (位置, 速度)。
        """
```

以下可見初始設定為：

- $\gamma = 0.99$
- 狀態 S：汽車的位置（[−1.2,0.6]）和速度（[−0.07,0.07]）被離散化為網格（預設 15x15）。
- 動作 A：三個離散動作影響汽車的加速度。

```python
        # discount factor for future rewards, typically close to 1,
        # to prioritize long-term rewards.
        self.gamma = 0.99

        # Discretization levels for position and velocity, defining the state space.
        self.discretization = discretization
        self.discretization_position = discretization[0]
        self.discretization_velocity = discretization[1]

        self.states = {(i, j) for i in range(self.discretization_position) for j in range(self.discretization_velocity)}

        # In MDP, actions A(s) are available choices at each state
        self.actions = [0, 1, 2]
```

```python
def sample_position_from_discretized(position, n):
    space = np.linspace(-1.2, 0.6, n)
    if position==n-1:
        return np.random.uniform(space[position], space[position]+0.1)

    return np.random.uniform(space[position], space[position+1])

def sample_velocity_from_discretized(velocity, n):
    space = np.linspace(-0.07, 0.07, n)
    if velocity==n-1:
        return np.random.uniform(space[velocity], space[velocity]+0.01)
    return np.random.uniform(space[velocity], space[velocity+1])
```

- compute_rewards：默認獎勵 R = -1，求快速到達目標（位置 ≥ 0.5 時獎勵為 0）。

```python
    def compute_rewards(self):
        """
        計算MDP的獎勵函數 R(s,s')。

        在MDP中,表示從 s 轉移到 s' 的即時獎勵。
        在Mountain Car問題中(課程05: 動態規劃):
        - 默認獎勵為 -1,懲罰每一步以鼓勵快速到達目標。
        - 當轉移到目標狀態(位置 >= 0.5)時,獎勵為 0,表示成功。

        返回:
            defaultdict: 映射 (state, new_state) 到獎勵值。
        """
        rewards = defaultdict(lambda: -1)

        for state in self.states:
            for action in self.actions:
                for new_state in self.states:
                    # Check if new_state corresponds to position >= 0.5.
                    # Convert discretized position to continuous to compare with goal.
                    position = sample_position_from_discretized(new_state[0], self.discretization_position)
                    if position >= 0.5:
                        rewards[(state, new_state)] = 0
        return rewards
```

- compute_transition_probabilities：基於物理位置和速度，通過採樣近似為概率轉移 P。

```python
    def compute_transition_probabilities(self):
        """
        計算MDP的轉移概率 P(s'|s,a)。

        在MDP中,表示從狀態 s 採取動作 a 後到達狀態 s' 的概率。
        Mountain Car的連續動態通過以下方式近似:
        1. 在每個離散區間內採樣連續狀態。
        2. 應用狀態轉移模型
        3. 離散化結果狀態並累積概率。

        返回:
            defaultdict: 映射 (state, action, new_state) 到概率值。
        """
        transition_probabilities = defaultdict(lambda: 0)
        for state in self.states:
            discrete_position, discrete_velocity = state
            for action in self.actions:
                for _ in range(NUMBER_OF_SAMPLES):
                    continuous_position = sample_position_from_discretized(discrete_position, self.discretization_position)
                    continuous_velocity = sample_velocity_from_discretized(discrete_velocity, self.discretization_velocity)

                    new_velocity = compute_new_velocity(continuous_position, continuous_velocity, action)
                    new_position = compute_new_position(continuous_position, new_velocity)

                    new_discrete_state = discretize(new_position, new_velocity, self.discretization)

                    #transition_probabilities[(state, action, new_discrete_state)] -= 1 / NUMBER_OF_SAMPLES
                    transition_probabilities[(state, action, new_discrete_state)] += 1 / NUMBER_OF_SAMPLES
        return transition_probabilities
```

- get_policy: 找到策略$\pi(s)$，將期望累積折扣獎勵最大化。

```python
def get_policy(mdp, V):
    """
    從價值函數 V*(s) 導出最佳策略 π*(s)。

    在MDP中,最佳策略 π*(s) 選擇最大化期望累積獎勵的動作(課程05):
        π*(s) = argmax_a Σ_{s'} P(s'|s,a) [ R(s,s') + γ V*(s') ]
    給定 V*(s),此貪婪策略是最優的。

    參數:
        mdp: 包含狀態、動作、轉移、獎勵和 γ 的MDP對象。
        V: 最佳價值函數,映射狀態到價值。

    返回:
        dict: 映射狀態到最佳動作。
    """
    policy = {}
    for state in mdp.states:
        policy[state] = max(mdp.actions, key=lambda action:
            sum(mdp.transition_probabilities.get((state, action, new_state), 0) *
                (mdp.rewards.get((state, new_state), -1) + mdp.gamma * V[new_state])
                # in new_state for mdp.states))
                for new_state in mdp.states))
    return policy
```

- value_iteration: 迭代計算價值函數 V(s)，表示從狀態 s 開始的期望累積獎勵，並通過選擇最大化期望回報的動作導出最佳策略。

```python
def value_iteration(mdp, num_iterations=NUM_ITERATIONS):
    """
    執行價值迭代,計算最佳價值函數 V*(s)。

    價值迭代(課程05: 動態規劃)通過迭代應用貝爾曼最優性方程估計 V*(s),
    表示從狀態 s 開始的最大期望累積獎勵:
        V_{k+1}(s) = max_a Σ_{s'} P(s'|s,a) [ R(s,s') + γ V_k(s') ]
    當 k → ∞ 時,收斂至 V*(s),為最佳策略提供基礎。

    參數:
        mdp: 包含狀態、動作、轉移、獎勵和 γ 的MDP對象。
        num_iterations: 迭代次數。

    返回:
        dict: 映射狀態到最佳價值 V*(s)。
    """
    V = {state: 0 for state in mdp.states}
    for _ in tqdm.tqdm(range(num_iterations)):
        for state in mdp.states:
            V[state] = max(
                sum(mdp.transition_probabilities.get((state, action, new_state), 0) *
                    (mdp.rewards.get((state, new_state), -1) + mdp.gamma * V[new_state])
                    # in new_state for mdp.states
                    for new_state in mdp.states)
                # in action for mdp.actions
                for action in mdp.actions
            )
    return V
```

1.3. Provide a thorough explanation of how the value iteration algorithm works. This explanation can be supported by either a diagram or a step-by-step illustrative algorithm.

(1) Value iteration 概述

Value iteration 目的為找到最佳 policy π，屬於 Dynamic programming (DP)。基於 Bellman optimally equation，從每個狀態 s 開始迭代計算，同步更新(synchronous backup) 逐步趨近最佳 V*(s)，導出最佳 policy π。

(2) 原理

若知道 subproblems V*(s’)，便可用以下公式（圖），計算當前(one-step lookahead)狀態的最佳 V*(s)

$$\boxed{v_*(s) \leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')}$$

講義 CH5 p.36

其中：

R_s^a為 Reward，在狀態 s 採取動作 a 之即時獎勵。

$P_{ss'}^a$為 Transition Probability，在狀態 s 採取動作 a 後，轉移到狀態 s’之機率。

(3) step-by-step illustrative algorithm

input：MDP (S, A, P, R, γ)

output：Find V*(s) and $\pi^*(s)$

1. $\mathrm{V(s)} \Leftarrow 0, \forall s \in S;$
2. $while\ \Delta \geq \Delta_0\ do$

    $\Delta \leftarrow 0;$

    $For\ each\ s \in S\ do$

    $temp \Leftarrow V(s);$

    $V(s) \leftarrow max_a \Sigma_{s'} P(s,a,s')[R(s,a,s') + \gamma V(s')]$

    $\Delta \leftarrow \max(\Delta, |\mathrm{temp} - \mathrm{V(s)}|)\,;$

    end

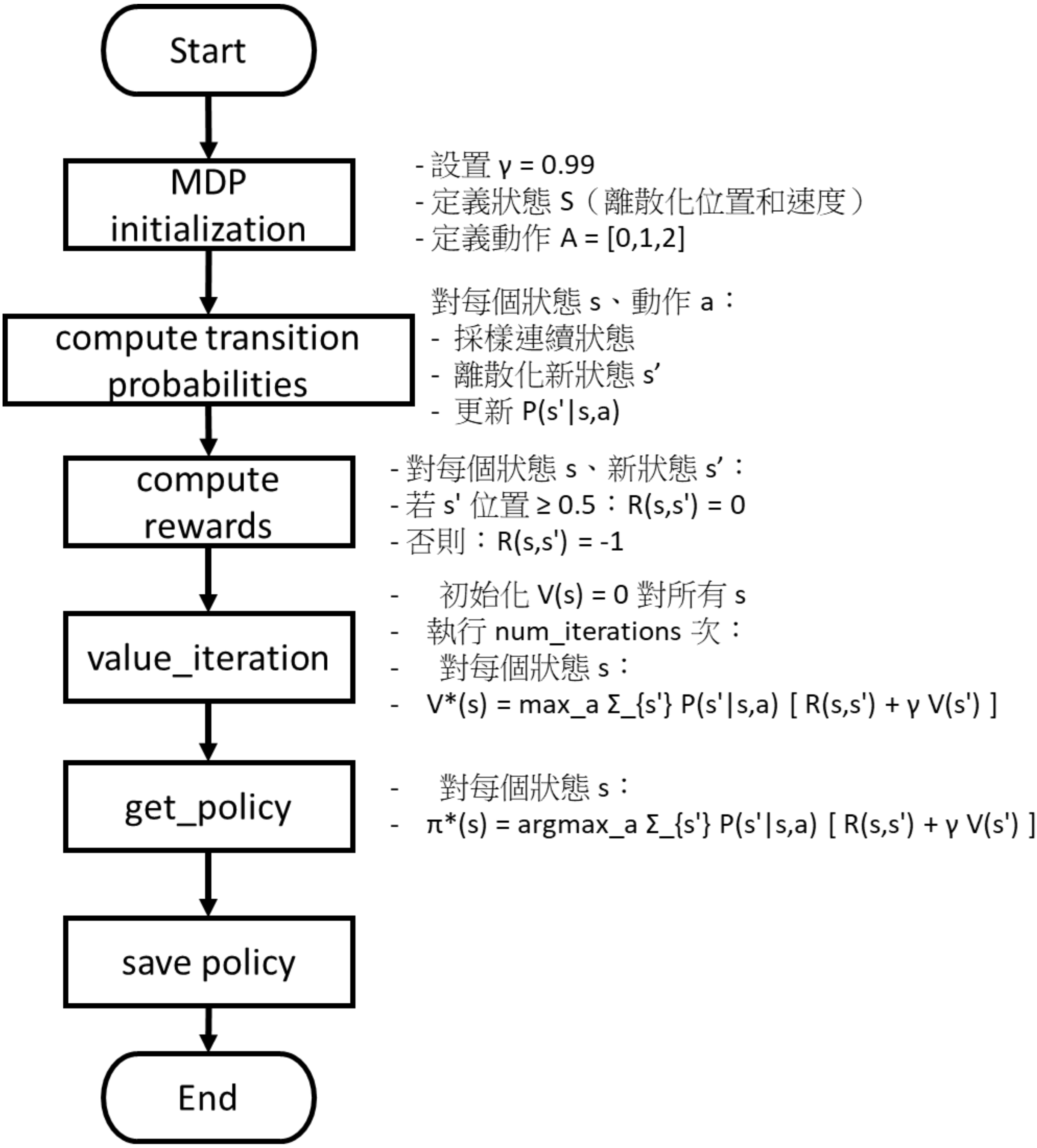
    end

    $\pi^*(\mathrm{s}) \leftarrow max_a\ \Sigma_{s'}\ P(s,a,s')[R(s,a,s') + \gamma V(s')], \forall s \in S;$

    return $\pi^*(\mathrm{s}), \forall s \in S$

ref: https://core-robotics.gatech.edu/2021/01/19/bootcamp-summer-2020-week-3-value-iteration-and-q-learning/

1.4. Create a flowchart that visually represents the overall structure and process flow of the codes.



1.5. Implement at least three meaningful changes to the code to demonstrate different outcomes. Higher credit will be grated for changes that align well with MDP or Dynamic Programming theory.

(1) 調整 discount value

初始程式碼設定 $\gamma = 0.99$，平均在 150 步前到達目標。

```
PS C:\Nora\上課\0414 智慧型控制\Mid takehome 2025> & "C:/Users/Nora Teng/AppData/Local/Programs/Python/Python310/python.exe" "c:/Nora/上課/0414 智慧型控制/M
id takehome 2025/mountain_car.py"
 73%|                    | 146/200 [00:04<00:01, 29.65it/s]
 72%|                    | 145/200 [00:04<00:01, 29.68it/s]
 72%|                    | 145/200 [00:04<00:01, 29.71it/s]
 72%|                    | 144/200 [00:04<00:01, 29.68it/s]
 72%|                    | 144/200 [00:04<00:01, 29.65it/s]
 74%|                    | 148/200 [00:04<00:01, 29.67it/s]
 74%|                    | 148/200 [00:04<00:01, 29.66it/s]
 62%|                    | 125/200 [00:04<00:02, 29.60it/s]
 72%|                    | 145/200 [00:04<00:01, 29.66it/s]
 74%|                    | 147/200 [00:04<00:01, 29.75it/s]
```

嘗試改為 $\gamma = 0.95$，可明顯看出效率變低，且部分嘗試可能沒有成功達到目標。

```
PS C:\Nora\上課\0414 智慧型控制\Mid takehome 2025> & "C:/Users/Nora Teng/AppData/Local/Programs/Python/Python310/python.exe" "c:/Nora/上課/0414 智慧型控制/M
id takehome 2025/mountain_car.py"
100%|                    | 199/200 [00:06<00:00, 29.83it/s]
100%|                    | 199/200 [00:06<00:00, 29.80it/s]
100%|                    | 199/200 [00:06<00:00, 29.81it/s]
 94%|                    | 187/200 [00:06<00:00, 29.86it/s]
 68%|                    | 137/200 [00:04<00:02, 29.77it/s]
 70%|                    | 139/200 [00:04<00:02, 29.69it/s]
100%|                    | 199/200 [00:06<00:00, 29.82it/s]
100%|                    | 199/200 [00:06<00:00, 29.82it/s]
100%|                    | 199/200 [00:06<00:00, 29.81it/s]
 92%|                    | 185/200 [00:06<00:00, 29.86it/s]
```

由以上對比可看出，$\gamma = 0.99$使策略更有效、更穩定，更接近。這表示更高的 γ 使策略更重視長期獎勵，能更快找到到達目標的路徑。

(2) 修改 reward function

原本的目標狀態獎勵為 0，其他為 -1，更改為：

- 目標狀態（position >= 0.5）獎勵為 +10。提高達到目標的獎勵（+10），更加激勵代理到達終點。
- 非目標狀態獎勵為 -1 - 0.5 * abs(velocity)，增加對速度的懲罰。促進代理在爬坡時控制速度，可減少無效的高速擺動。

```python
def compute_rewards(self):
    rewards = defaultdict(lambda: -1)
    for state in self.states:
        for action in self.actions:
            for new_state in self.states:
                position = sample_position_from_discretized(new_state[0], self.discretization_position)
                velocity = sample_velocity_from_discretized(new_state[1], self.discretization_velocity)
                if position >= 0.5:
                    rewards[(state, new_state)] = 10
                else:
                    rewards[(state, new_state)] = -1 - 0.5 * abs(velocity)
    return rewards
```

結果如下：

```
PS C:\Nora\上課\0414 智慧型控制\Mid takehome 2025> & "C:/Users/Nora Teng/AppData/Local/Programs/Python/Python310/python.exe" "c:/Nora/上課/0414 智慧型控制/M
id takehome 2025/mountain_car.py"
 64%|          | 129/200 [00:04<00:02, 29.76it/s]
 60%|          | 120/200 [00:04<00:02, 29.69it/s]
 76%|          | 152/200 [00:05<00:01, 29.67it/s]
 65%|          | 130/200 [00:04<00:02, 29.70it/s]
 67%|          | 134/200 [00:04<00:02, 29.67it/s]
 83%|          | 166/200 [00:05<00:01, 29.81it/s]
 76%|          | 152/200 [00:05<00:01, 29.76it/s]
 87%|          | 174/200 [00:05<00:00, 29.74it/s]
 83%|          | 166/200 [00:05<00:01, 29.78it/s]
 64%|          | 128/200 [00:04<00:02, 29.69it/s]
```

平均比 $\gamma = 0.99$更為快速完成目標，可見將獎勵提高可以提升效率。

(3) 修改 value_iteration function

在 value_iteration 中新增參數 theta=0.01，並在每次迭代計算值函數的最大變化量 delta：若 delta < theta，則提前終止迭代。

此作法屬於 DP，通過反覆更新值函數直至收斂，再加入收斂檢查減少不必要的迭代次數，當 value 變化很小後便停止計算，確保 value 在接近最優時停止，從而提高效率，符合 MDP 核心理論。

```python
def value_iteration(mdp, num_iterations=NUM_ITERATIONS, theta=0.01):
    V = {state: 0 for state in mdp.states}
    for _ in tqdm.tqdm(range(num_iterations)):
        delta = 0
        for state in mdp.states:
            v = V[state]
            V[state] = max(
                sum(mdp.transition_probabilities.get((state, action, new_state), 0) *
                    (mdp.rewards.get((state, new_state), -1) + mdp.gamma * V[new_state])
                    for new_state in mdp.states)
                for action in mdp.actions
            )
            delta = max(delta, abs(v - V[state]))
        # Change 3: Added convergence check for early stopping
        if delta < theta:
            print(f"Value iteration converged after {_ + 1} iterations")
            break
    return V
```

```
PS C:\Nora\上課\0414 智慧型控制\Mid takehome 2025> & "C:/Users/Nora Teng/AppData/Local/Programs/Python/Python310/python.exe" "c:/Nora/上課/0414 智慧型控制/M
id takehome 2025/mountain_car.py"
 48%|          |  96/200 [00:03<00:03, 29.75it/s]
 58%|          | 116/200 [00:03<00:02, 29.71it/s]
 53%|          | 106/200 [00:03<00:03, 29.74it/s]
 53%|          | 106/200 [00:03<00:03, 29.68it/s]
 57%|          | 114/200 [00:03<00:02, 29.63it/s]
 51%|          | 102/200 [00:03<00:03, 29.72it/s]
 57%|          | 114/200 [00:03<00:02, 29.68it/s]
 57%|          | 115/200 [00:03<00:02, 29.66it/s]
 56%|          | 113/200 [00:03<00:02, 29.56it/s]
 48%|          |  96/200 [00:03<00:03, 29.67it/s]
```

可看出 value_iteration 在三種嘗試中是效率最高的。

2. (20%) Regarding Bellman equation which is a crucial necessary condition for optimality associated with the optimization of dynamic programming method:

2.1. Based on the concepts introduced in Lesson 04, derive the Bellman expectation equations by hand writing, showing the step-by-step formulation process.

2.1 Derive Bellman expectation equation

p29. state-value func.

$$V_\pi(s) = E_\pi[G_t \mid S_t = s]$$

$$= \sum_{a\in A} \pi(a|s) \underbrace{E_\pi[G_t \mid S_t = s, A_t = a]}_{\text{action-value func.}}$$

$$= \sum_{a\in A} \pi(a|s)\, q_\pi(s,a) \quad \text{— ①}$$

$$q_\pi(s,a) = E_\pi[G_t \mid S_t = s, A_t = a] \quad , \quad G_t = R_{t+1} + \gamma G_{t+1}$$

$$= E_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \quad \text{— ②}$$

又 $\begin{cases} E_\pi[R_{t+1} \mid S_t = s, A_t = a] = R_s^a \\ E_\pi[G_{t+1} \mid S_t = s, A_t = a] = \sum_{s'\in S} P_{ss'}^a V_\pi(s') \end{cases}$ 代回 ②

$$\Rightarrow q_\pi(s,a) = R_{ss'}^a + \gamma \sum_{s\in S} P_{ss'}^a V_\pi(s') \quad \text{— ③} \qquad \text{代回 ①}$$

$$\Rightarrow V_\pi(s) = \sum_{a\in A} \pi(a|s) \Big[\underbrace{R_{ss'}^a + \gamma \sum_{s\in S} P_{ss'}^a V_\pi(s')}_{q_\pi(s,a)}\Big] \qquad \text{代回 ③}$$

$$\Rightarrow q_\pi(s,a) = R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a \underbrace{\sum_{a'\in A} \pi(a'|s')\, q_\pi(s',a')}_{V_\pi(s)} \qquad \#$$

2.2. Using the same notation style, formulate the Bellman optimality equations by hand writing. Additionally, provide a rationale for why these equations can be used to reliably arrive the optimal policy.

2.2 Bellman optimally equation

$$\begin{cases} V_\pi(s) = \sum_{a \in A} \pi(a|s)\, q_\pi(s,a) \\ q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_\pi(s') \end{cases}$$ from 2.1

Define optimal state function

$$\Rightarrow V^*(s) = \max_\pi V_\pi(s)$$

Define optimal action-value function

$$\Rightarrow q^*(s,a) = R_s^a + r \sum_{s' \in S} P_{ss'}^a V^*(s')$$

$$V^*(s) = \max_{a \in A} q^*(s,a)$$

$$= \max_{a \in A} \left(R_s^a + r \sum_{s' \in S} P_{ss'}^a V^*(s) \right) \#$$

$$\Rightarrow q^*(s,a) = R_s^a + r \sum_{s' \in S} P_{ss'}^a \max_{a' \in A} q^*(s',a') \#$$

3. (30%) Referring to the Jack’s Car Rental scenario from Homework 2,

3.1. Describe the reasoning behind using the Poisson distribution to model the probabilities of car returns and rental requests.

Poisson 分佈適合描述單位時間（或空間）內隨機事件發生的次數。事件需要保持恆定事件間歇發生的速率。而本題 Jack’s Car Rental 滿足條件如以下：

- 隨機性：汽車的歸還與租借在一天中隨機發生，無法完全預測。
- 單位時間內事件次數：Poisson 分布適合描述在固定時間內發生某事件的次數，如每小時有多少人來租車或還車。
- 事件獨立性：每次租借或歸還事件彼此獨立，符合 Poisson 分布的假設條件。

因此，Poisson 分布能合理地反映 Jack’s Car Rental 這種隨機、獨立且分散發生的事件的機率。

3.2. Update your implementation to reflect the following new scenario: - One of Jack’s employees at location 1 commutes home by bus and lives close to location 2. This employee is willing to transfer one car to location 2 each night at no cost. All other car transfers, including those in the opposite direction, still cost 2 dollars per vehicle. Additionally, Jack now faces a parking constraint: If more than 10 cars are kept overnight at a location (after any moving of cars), then an additional cost of 4 dollars must be incurred to use a secondary parking reservation (independent of how many cars are kept in the reservation).

Updating codes:

- Transferring one car to location 2 each night at no cost. In the opposite direction, still cost 2 dollars per vehicle.
    - a >= 1（從地點 1 到地點 2 轉移），則第一輛車免費，其餘車輛每輛 2 美元：move_cost = -(a - 1) * 2 if a > 1 else 0。
    - a < 0（從地點 2 到地點 1 轉移），則仍需支付每輛車 2 美元：move_cost = -math.fabs(a) *

```python
    def get_transition_model(self, s, a):
        # 調整汽車轉移：員工免費轉移 1 輛車從地點 1 到地點 2
        effective_a = a  # 實際需要計算費用的轉移數量
        if a >= 1:
            # 如果從地點 1 轉移到地點 2 的數量 >= 1，則 1 輛車免費
            move_cost = -(a - 1) * 2 if a > 1 else 0  # 免費轉移 1 輛，其餘每輛 2 美元
        else:
            # 從地點 2 到地點 1 的轉移仍需支付每輛 2 美元
            move_cost = -math.fabs(a) * 2
```

● 汽車轉移後，檢查每個地點的汽車數量 s = (s[0] - a, s[1] + a)。

```python
        # 執行汽車轉移
        s = (s[0] - a, s[1] + a)  # 移動 a 輛車從地點 1 到地點 2
```

● If more than 10 cars are kept overnight at a location (after any moving of cars), then an additional cost of 4 dollars.
if loc_cars > 10: parking_cost -= 4。

```python
        # 計算停車費用：如果過夜汽車數量超過 10 輛，則每個地點額外支付 4 美元
        parking_cost = 0
        for loc_cars in s:
            if loc_cars > 10:
                parking_cost -= 4  # 每個地點超過 10 輛支付 4 美元

        # 計算總移動和停車費用
        move_reward = move_cost + parking_cost
```

● 最後計算總移動和費用，不須更動

```python
        # 計算總移動和停車費用
        move_reward = move_cost + parking_cost

        t_prob, expected_r = ([{}, {}], [{}, {}])
        for loc in range(2):
            morning_cars = s[loc]
            rent_return_pmf = self.rent_return_pmf[loc]
            for rents in range(morning_cars + 1):
                max_returns = self.max_cars - morning_cars + rents
                for returns in range(max_returns + 1):
                    p = rent_return_pmf[morning_cars, rents, returns]
                    if p < 1e-5:
                        continue
                    s_prime = morning_cars - rents + returns
                    r = rents * 10  # 每輛車租賃收入 10 美元
                    t_prob[loc][s_prime] = t_prob[loc].get(s_prime, 0) + p
                    expected_r[loc][s_prime] = expected_r[loc].get(s_prime, 0) + p * r
```

```python
        # 合併兩個地點的概率和期望回報
        t_model, r_model = ({}, {})
        for s_prime1 in t_prob[0]:
            for s_prime2 in t_prob[1]:
                p1 = t_prob[0][s_prime1]  # 地點 1 的 p(s' | s, a)
                p2 = t_prob[1][s_prime2]  # 地點 2 的 p(s' | s, a)
                t_model[(s_prime1, s_prime2)] = p1 * p2
                # 計算期望回報，需標準化
                norm_E1 = expected_r[0][s_prime1] / p1
                norm_E2 = expected_r[1][s_prime2] / p2
                r_model[(s_prime1, s_prime2)] = norm_E1 + norm_E2 + move_reward

        return t_model, r_model
```

- Result comparison

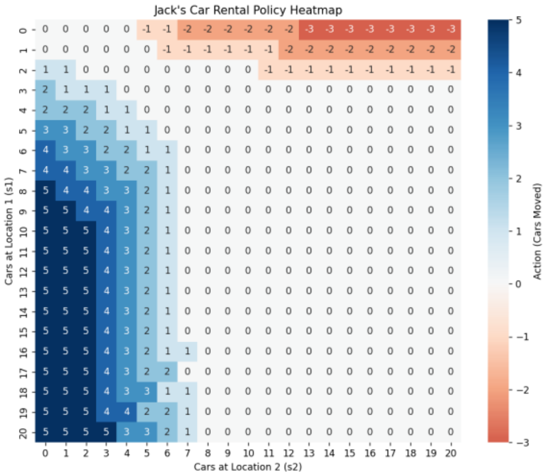
(1) HW2

HW2 條件為：

移動費用：每輛車移動 2 美元（無免費轉移）。

```
Iteration 30: max delta = 0.44

Value iteration done, final policy:
[[ 0  0  0  0  0 -1 -1 -2 -2 -2 -2 -2 -2 -3 -3 -3 -3 -3 -3 -3 -3]
 [ 0  0  0  0  0  0 -1 -1 -1 -1 -1 -1 -2 -2 -2 -2 -2 -2 -2 -2 -2]
 [ 1  1  0  0  0  0  0  0  0  0  0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
 [ 2  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 2  2  2  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 3  3  2  2  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 4  3  3  2  2  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  4  4  3  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  2  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  3  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  4  2  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  5  3  3  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0]]
```



由右側熱力圖可看出，左下角多是正值（loc1 有車多，會往 loc2 移動） 右上角多是負值（loc2 有車多，會往 loc1 移動），而圖片中間很快就變為 0，代表在大部分狀態下移車是無益處的。可驗證 HW2 之策略較規律、平滑，對應簡單規則。

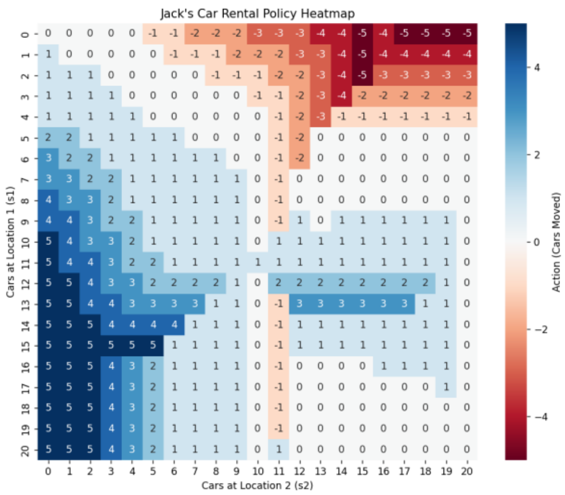
(2) Midterm

條件為：

- 移動費用：地點 1 到地點 2 的第一輛車免費，其餘車輛每輛 2 美元；地點 2 到地點 1 的轉移仍為每輛 2 美元。
- 停車限制：如果某地點過夜汽車數量超過 10 輛，需支付額外 4 美元的停車費用。

```
Iteration 30: max delta = 0.45

Value iteration done, final policy:
[[ 0  0  0  0  0 -1 -1 -2 -2 -2 -3 -3 -3 -4 -4 -5 -4 -5 -5 -5 -5]
 [ 1  0  0  0  0  0 -1 -1 -1 -2 -2 -2 -3 -3 -4 -5 -4 -4 -4 -4 -4]
 [ 1  1  1  0  0  0  0  0 -1 -1 -1 -2 -2 -3 -4 -5 -3 -3 -3 -3 -3]
 [ 1  1  1  1  0  0  0  0  0  0 -1 -1 -2 -3 -4 -2 -2 -2 -2 -2 -2]
 [ 1  1  1  1  1  0  0  0  0  0  0 -1 -2 -3 -1 -1 -1 -1 -1 -1 -1]
 [ 2  2  1  1  1  1  1  0  0  0  0 -1 -2  0  0  0  0  0  0  0  0]
 [ 3  2  2  1  1  1  1  1  1  0  0 -1 -2  0  0  0  0  0  0  0  0]
 [ 3  3  2  2  1  1  1  1  1  1  0 -1  0  0  0  0  0  0  0  0  0]
 [ 4  3  3  2  1  1  1  1  1  1  0 -1  0  0  0  0  0  0  0  0  0]
 [ 4  4  3  2  2  1  1  1  1  1  0 -1  1  0  1  1  1  1  1  1  0]
 [ 5  4  3  3  2  1  1  1  1  1  0  1  1  1  1  1  1  1  1  1  0]
 [ 5  4  4  3  2  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  0]
 [ 5  5  4  3  3  2  2  2  2  1  0  2  2  2  2  2  2  2  2  1  0]
 [ 5  5  4  4  3  3  3  3  1  1  0 -1  3  3  3  3  3  3  1  1  0]
 [ 5  5  5  4  4  4  4  1  1  1  0 -1  1  1  1  1  1  1  1  1  0]
 [ 5  5  5  5  5  5  1  1  1  1  0 -1  1  1  1  1  1  1  1  1  0]
 [ 5  5  5  4  3  2  1  1  1  1  0 -1  0  0  0  0  1  1  1  1  0]
 [ 5  5  5  4  3  2  1  1  1  1  0 -1  0  0  0  0  0  0  0  1  0]
 [ 5  5  5  4  3  2  1  1  1  1  0 -1  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  1  1  1  1  0 -1  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  1  1  1  1  0  1  0  0  0  0  0  0  0  0  0]]
```



觀察熱力圖可發現非 0 的區塊比 HW2 多很多，且多為調 1 輛車到地點 2，對應第一輛車免費之策略、對收益最有利。

此策略會避免讓任一地點車輛超過 10 輛。對比兩張熱力圖中間欄位可看出，原本大量 0，出現了 -1、-2（將車從該地點移開）。其表示在地點車輛數過多時，會主動轉移以避免需付停車費。

相較於 HW2 簡單的策略，現在的策略在中段出現更多小變動，也就對應不同條件下的不同成本：免費移動 v.s. 停車費用。