中山大学 DCS5706《随机过程及应用》期末作业

$M/M/1$ 排队系统的控制研究

何瑞杰 25110801

目录

1. 问题描述

$M/M/1$ 排队系统广泛存在于生产生活中，它指的是一个先到先服务的单服务台的服务系统。顾客按照参数为 λ 的 Poisson 过程到达；服务台的服务时间服从参数为 μ（即服务速率）的指数分布，且和顾客的到达过程独立。

现考虑带有服务速率控制的 $M/M/1$ 排队系统，其服务速率 $\mu(i) \in (\lambda, \bar{\mu}]$ 取决于系统中的顾客数目 i，该数目包括等待的顾客和正在服务的顾客。系统有两重成本：第一重为单位时间的服务成本 $q(\mu)$，其满足 $q(0) = 0$；第二重为顾客等待成本 $c(i)$。对该 $M/M/1$ 系统的控制目标为对系统内不同顾客数量时采用不同服务速率，以期最小化单位总成本。

2. 模型建立

带有控制的 $M/M/1$ 排队系统可使用连续时间 Markov 决策过程建模，其各参数如下：

CTMDP 资料	$M/M/1$ 系统中的元素
状态 $x(t)$	系统中该时刻的顾客数目 i
动作 $u(t)$	系统该时刻的服务速率 μ
代价函数 $g(x(t), u(t))$	单位时间总成本 $q(\mu) + c(i)$
策略 μ_k	系统的服务速度策略 $\mu(i)$

若转移速度对所有状态和动作均匀，则有

$$J_\pi(x_0) = \mathbb{E}\left[\sum_{k=0}^{\infty}\left(\frac{\nu}{\beta+\nu}\right)^k \frac{g(x_k, \mu_k(x_k))}{\beta+\nu}\right] = \mathbb{E}\left[\sum_{k=0}^{\infty}\alpha^k \cdot \tilde{g}(x_k, \mu_k(x_k))\right],$$

对应的 Bellman 方程为

$$J(i) = \frac{1}{\beta+\nu} \min_{u\in U(i)} \left[g(i,u) + \nu \sum_j p_{i,j}(u) J(j) \right]$$

考虑转移速度对所有状态和动作不均匀, 但存在上界 ν, 若对状态 i 和动作 u, 有转移速度 $\nu_i(u)$, 考虑下面拥有新的转移概率的均匀转移速度的 CTMDP:

$$\tilde{p}_{i,j} = \begin{cases} \dfrac{\nu_i(u)}{\nu} p_{i,j}(u) & \text{if } i \neq j \\ \dfrac{\nu_i(u)}{\nu} p_{i,i}(u) + 1 - \dfrac{\nu_i(u)}{\nu} & \text{if } i = j \end{cases}$$

因此新的 CTMDP 的 Bellman 方程为

$$J(i) = \frac{1}{\beta+\nu} \min_{u\in U(i)} \left[g(i,u) + (\nu - \nu_i(u))J(i) + \nu_i(u) \sum_j p_{i,j}(u) J(j) \right]$$

在 $M/M/1$ 队列中, 转移速率 $\nu_i(\mu)$ 在系统中无顾客 ($i=0$) 时为 λ, 在有顾客时为 $\lambda+\mu$, 则依照上述结果的转移速率上界为 $\nu = \lambda + \bar{\mu}$。由于该系统的状态只可能向相邻状态转移, 且当系统中没有顾客时, 规定 $\mu(0)=0$, 因此可以得到其 Bellman 方程为

$$J(i) = \begin{cases} \dfrac{1}{\beta+\nu}[c(0) + (\nu-\lambda)J(0) + \lambda J(1)] & i = 0 \\ \dfrac{1}{\beta+\nu} \min\limits_{\mu}[c(i) + q(\mu) + (\nu-\lambda-\mu)J(i) + \lambda J(i+1) + \mu J(i-1)] & 1 \leqslant i < M \\ \dfrac{1}{\beta+\nu} \min\limits_{\mu}[c(M) + q(\mu) + (\nu-\mu)J(M) + \mu J(M-1)] & i = M \end{cases}$$

注意系统中转移概率 $p_{i,i+1}(u)$ 对应着新顾客进入系统, 其值为 $\dfrac{\lambda}{\lambda+\mu}$, 而 $p_{i,i-1}(u)$ 对应着顾客服务完成离开系统, 其值为 $\dfrac{\mu}{\lambda+\mu}$。

3. 最优策略计算

本节介绍代价函数的取法和求解 Bellman 方程用到的算法。

3.1. 代价函数

本项目研究排队代价和服务代价分别为线性、二次函数、指数函数情况时的最优控制策略, 共有九种组合。具体地, 线性、二次代价和指数代价分别取

$$f_{\text{linear}}(x) = x,$$
$$f_{\text{quad}}(x) = \frac{1}{2}x^2,$$
$$f_{\text{exp}}(x) = e^{0.1x}.$$

3.2. 值迭代

第一种求解方法是值迭代, 其原理为直接应用 Bellman 方程的定义, 并用其迭代让边界处的值逐渐传导到其他各个状态, 直至收敛:

$$J_{k+1}(i) = \min_{u\in U(i)} \left[g(i,u) + \sum_{j=1}^{n} p_{i,j}(u) J_k(j) \right]$$

在 $M/M/1$ 系统中，值迭代算法可以写为

Algorithm 1 Value Iteration for Controlled M/M/1 Queue

```
procedure Value-Iteration($c(i)$, $q(\mu)$, $\lambda$, $\bar{\mu}$, $\beta$, $\varepsilon$)
    $\nu \leftarrow \lambda + \bar{\mu}$
    $J_0(i) \leftarrow 0, \forall i \in \{0, \ldots, N\}$
    $k \leftarrow 0$
    while true do
        $J_{k+1}(0) \leftarrow \dfrac{1}{\beta+\nu}\left[c(0) + (\nu-\lambda)J_{k(0)} + \lambda J_{k(1)}\right]$
        for $i \leftarrow 1, \ldots, N-1$ do
            $J_{k+1}(i) \leftarrow \frac{1}{\beta+\nu} \min_{\mu\in(\lambda,\bar{\mu}]}\left[c(i) + q(\mu) + \mu J_{k(i-1)} + (\nu-\lambda-\mu)J_{k(i)} + \lambda J_{k(i+1)}\right]$
        end
        $J_{k+1}(N) \leftarrow J_{k+1}(N-1)$
        if $\max_i |J_{k+1}(i) - J_{k(i)}| < \varepsilon$ then
            break
        end
        $k \leftarrow k+1$
    end
    $\mu^*(i) \leftarrow \arg\min_\mu [q(\mu) - \mu(J_{k+1}(i) - J_{k+1}(i-1))], \forall i \geq 1$
    return $(J_{k+1}, \mu^*)$
end
```

3.3. 策略迭代

还可以通过策略迭代算法解 Bellman 方程。其核心为从一个初始策略 $\mu^{(0)}$ 出发，每个迭代循环中，通过策略评估得到当前策略对应的价值函数 $J_{\mu^{(i-1)}}$，然后根据这个价值函数贪心地取得新的策略 $\mu^{(i)}$，直至策略收敛。

Algorithm 2 Policy Iteration for Controlled M/M/1 Queue

```
procedure Policy-Iteration($c(i)$, $q(\mu)$, $\lambda$, $\bar{\mu}$, $\beta$)
    $\mu_0(i) \leftarrow \bar{\mu}, \forall i \in \{1, \ldots, N\}$
    $k \leftarrow 0$
    while true do
        Solve linear system for $J_{\mu_k}$.
        for $i \leftarrow 1, \ldots, N$ do
            $\Delta J \leftarrow J_{\mu_k}(i) - J_{\mu_k}(i-1)$
            $\mu_{k+1}(i) \leftarrow \arg\min_{\mu\in(\lambda,\bar{\mu}]}[q(\mu) - \mu \cdot \Delta J]$
        end
        if $\mu_{k+1}(i) = \mu_{k(i)}, \forall i$ then
            break
        end
        $k \leftarrow k+1$
    end
    return $(J_{\mu_k}, \mu_k)$
end
```

3.4. 不同损失组合下的迭代结果

实际测试中，到达速率 $\lambda = 10$，系统最大容量 $N = 100$，折扣参数 $\beta = 0.01$。值迭代和策略迭代收敛到同样的策略，但值迭代相比策略迭代慢得多，因此这里使用策略迭代。通过最优策

略可以计算得到 Q 矩阵，进而计算出系统的稳态分布，最后得到系统的平均代价，实测这九个损失组合的最优策略的平均代价与实际模拟得出的代价基本一致（见附录）。

将不同损失组合下迭代得到的最优策略函数绘制如下，可见当服务损失时线性，而排队损失时线性、二次或指数时，最优策略在系统顾客数量较低时迅速增长至最大服务速率。当排队损失为线性时，不论服务损失时二次或是指数，最优策略随系统中顾客数量增长对应的系统服务速度增长较为缓慢。其他情况下，随系统中顾客数量的增长，最优策略下的服务速率先是立刻以大斜率增加，然后缓慢或线性增加，至系统中顾客人数在总容纳量一半左右时到达最高服务速率。

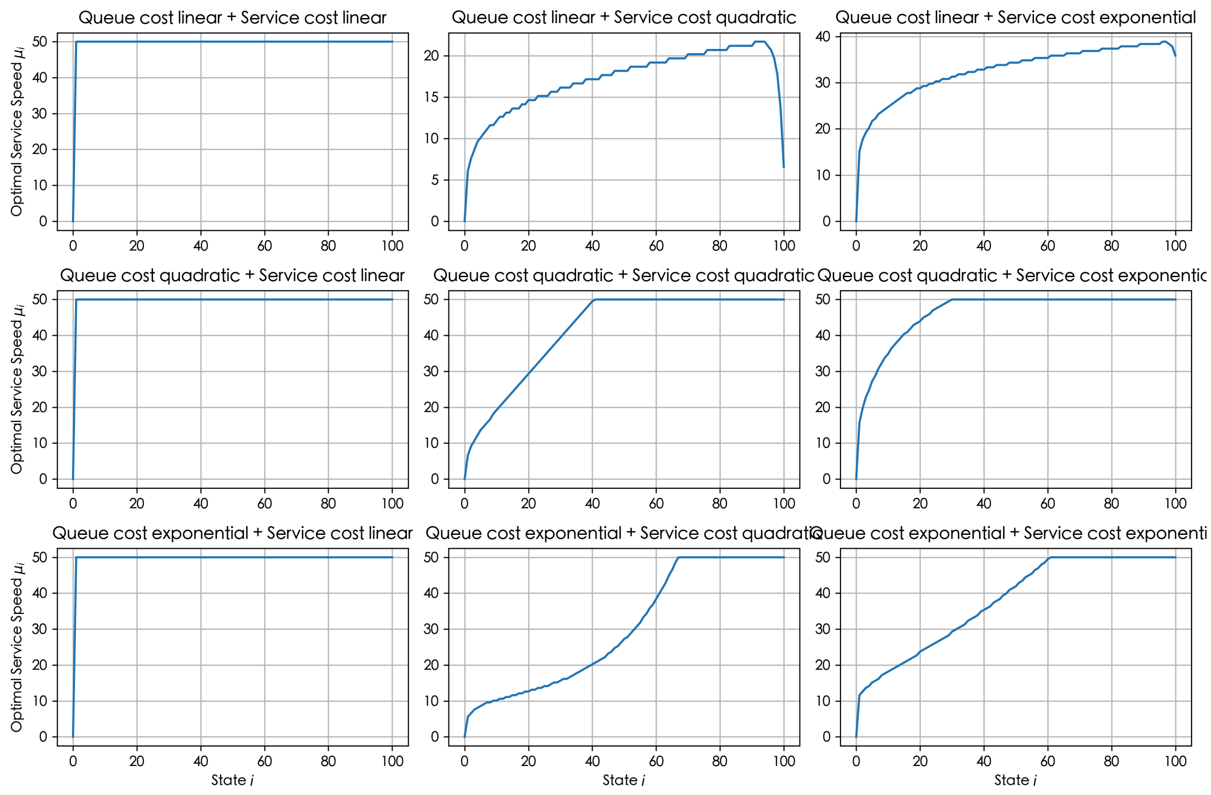

图 1　不同损失组合下的最优服务速度策略

3.5. $M/M/1$ 排队系统的仿真模拟

本节简述仿真模拟的逻辑。由于 CTMDP 的跳变总是发生在瞬间，除了跳变的时刻外其他时刻该过程的状态均在跳变间隔中恒定不变，因此可以采用离散事件法对该系统进行建模。具体地，将系统状态 i 表示为当前系统中顾客人数，将系统服务速度 μ_i 表示为当前系统服务速率，将系统到达速率 λ 表示为顾客到达系统的平均间隔时间的倒数，将系统最大容量 N 表示为系统最多可以容纳的顾客人数。

维护一个事件优先队列。仿真模拟开始时，系统中顾客人数为零，系统服务速率为零，并在队列中添加一个新的到达事件，事件距离系统当前时刻的间隔采样自到达过程的间隔分布。仿真系统处理完每个事件后，将会直接跳转至下一个事件的发生时刻。如果该事件是一个到达事件，系统中顾客数 $+1$，如果达前系统为空，系统开始服务该顾客，并在队列中添加一个服务完成时间的事件，即离开事件；如果到达前系统已满，则忽略该到达事件。如果该事件是离开事件，系统中顾客数 -1，如果离开后系统中顾客人数不为零，将在事件序列中添加新的离开事件；否则将系统服务速率降低至 0。在系统处理每一个事件时，都会计算自上一个事件以来的代价函数。如果事件队列中最近的事件发生时间超过了提前设定的仿真时长，系统终止，并根据仿真时长和总损失计算平均损失。

4. 模型参数和折扣参数对最优策略的影响

本节中固定排队损失为线性，服务损失为二次函数，研究系统参数中到达速率 λ、系统最大容量 N、折扣参数 β 对最优策略的影响。考虑 $\lambda \in \{5, 10, 20\}$，$N \in \{100, 1000, 2000\}$，$\beta \in \{0.001, 0.01, 0.1\}$，下图显示不同组合下的最优服务速度策略。我们发现一个有趣的情况。当 λ 较大或 β 较小时，系统的最优策略下服务速度随着系统内顾客数量增加而下降。

但 N 较大时，即使 λ 较大并不会产生上述情形，这说明系统的容量会显著影响最优策略。另外注意到当 β 较大时，即使增大系统容量，系统最优策略在大顾客量时下降的现象依然出现，这说明折扣参数会显著影响系统策略。

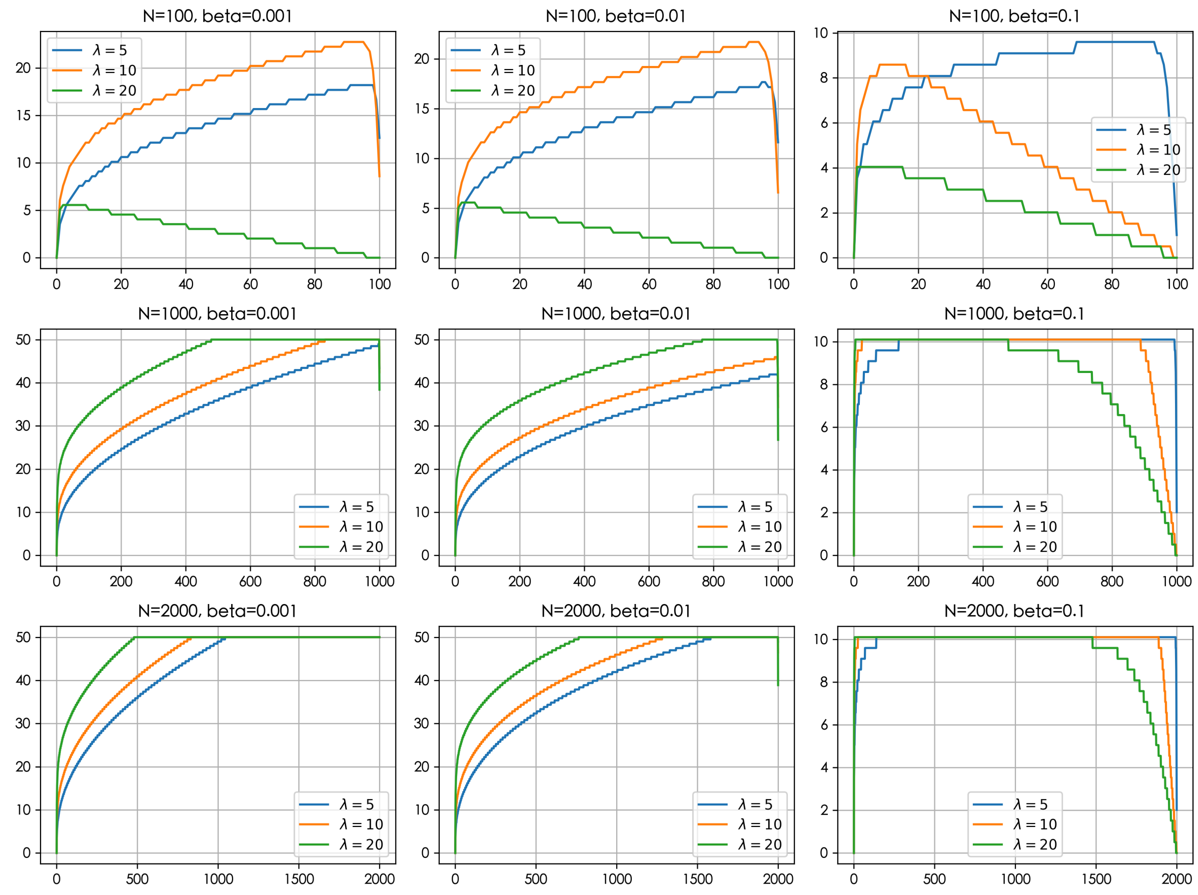

图 2　不同损失组合下的最优服务速度策略

5. 代码附录

5.1. $M/M/1$ 仿真模拟

Python

```python
# mm1.py
import heapq
import os
import numpy as np
from collections import namedtuple
from typing import Callable, List, Tuple, Optional
import matplotlib.pyplot as plt
import matplotlib as mpl
from tqdm import tqdm

plt.rcParams['font.family'] = list({"SimHei", "Heiti TC"} & set(f.name for f in
mpl.font_manager.fontManager.ttflist))[0]

Event = namedtuple('Event', ['time', 'type', 'data'])
# type: 'arrival', 'departure', 'rate_change'

class ControlledMM1Queue:

    def __init__(self,
                 arrival_rate: float,
                 service_rate_policy: Callable[[int], float],
                 service_cost_func: Callable[[float], float],
                 queue_cost_func: Callable[[int], float],
                 max_customers: int = None):
        self.arrival_rate = arrival_rate
        self.service_rate_policy = service_rate_policy
        self.service_cost_func = service_cost_func
        self.queue_cost_func = queue_cost_func
        self.max_customers = max_customers

        # system state
        self.current_time = 0.0
        self.num_customers = 0
        self.current_service_rate = 0.0

        # event priority queue
        self.event_queue: List[Event] = []

        # history
        self.history = {
            'time': [0.0],
            'num_customers': [0],
            'service_rate': [0.0],
            'total_cost': [0.0]
        }
        self.total_cost = 0.0
        self.last_event_time = 0.0

    def reset(self):
        self.current_time = 0.0
        self.num_customers = 0
        self.current_service_rate = 0.0
        self.event_queue = []
        self.history = {
            'time': [0.0],
            'num_customers': [0],
            'service_rate': [0.0],
            'total_cost': [0.0]
        }
        self.total_cost = 0.0
        self.last_event_time = 0.0

    def _add_event(self, time: float, event_type: str, data=None):
        heapq.heappush(self.event_queue, Event(time, event_type, data))

    def _exponential_sample(self, rate: float) -> float:
        if rate <= 0:
            return float('inf')
        return np.random.exponential(1.0 / rate)

    def _schedule_arrival(self):
        inter_arrival = self._exponential_sample(self.arrival_rate)
        next_arrival_time = self.current_time + inter_arrival
        self._add_event(next_arrival_time, 'arrival')

    def _schedule_departure(self):
        if self.num_customers > 0:
            # schedule next departure time
            service_rate = self.service_rate_policy(self.num_customers)
            service_time = self._exponential_sample(service_rate)
            next_departure_time = self.current_time + service_time
            self._add_event(next_departure_time, 'departure')
            self.current_service_rate = service_rate
        else:
            # system is empty
            self.current_service_rate = 0.0

    def _update_cost(self):
        dt = self.current_time - self.last_event_time
```

```python
        if dt > 0:
            # add period cost
            queue_cost = self.queue_cost_func(self.num_customers) * dt
            service_cost = self.service_cost_func(self.current_service_rate) * dt
            self.total_cost += queue_cost + service_cost

        # update history
        self.history['time'].append(self.current_time)
        self.history['num_customers'].append(self.num_customers)
        self.history['service_rate'].append(self.current_service_rate)
        self.history['total_cost'].append(self.total_cost)

        self.last_event_time = self.current_time

    def _handle_arrival(self):
        self._update_cost()

        # maximum capacity check
        if self.max_customers is None or self.num_customers < self.max_customers:
            self.num_customers += 1

            # start service when system turns to non-empty
            if self.num_customers == 1:
                self._schedule_departure()

        # schedule next arrival
        self._schedule_arrival()

    def _handle_departure(self):
        self._update_cost()

        if self.num_customers > 0:
            self.num_customers -= 1

            # serve next customer if system is non-empty
            if self.num_customers > 0:
                self._schedule_departure()
            else:
                self.current_service_rate = 0.0

    def run(self, T: float) -> dict:
        self.reset()

        self._schedule_arrival()

        while self.event_queue:
            # check to the nearest event
            event = heapq.heappop(self.event_queue)

            # check time limit
            if event.time > T:
                break

            # jump to the nearest event
            self.current_time = event.time

            # handle event
            if event.type == 'arrival':
                self._handle_arrival()
            elif event.type == 'departure':
                self._handle_departure()

            print(f"Simulating... {self.current_time/T*100:.2f}", end='\r')

        # update current time and cost
        self.current_time = T
        self._update_cost()

        return self.history

    def get_average_cost(self, T: float = None) -> float:
        if T is None:
            T = self.history['time'][-1]
        return self.total_cost / T
```

5.2. 策略迭代和价值迭代

```python
# solve_bellman.py

import numpy as np
from typing import Callable, List
import matplotlib.pyplot as plt
import matplotlib as mpl
from tqdm import tqdm
from scipy.sparse import diags
from scipy.sparse.linalg import spsolve

plt.rcParams['font.family'] = list({"SimHei", "Heiti TC"} & set(f.name for f in mpl.font_manager.fontManager.ttflist))[0]


class CTMDPControlledQueue:

    def __init__(self,
                 lambda_rate: float,
                 max_state: int,
                 c_func: Callable[[int], float],
                 q_func: Callable[[float], float],
                 mu_space: List[float],
                 beta: float = 0.1):
        self.lambda_rate = lambda_rate
        self.max_state = max_state
        self.c_func = c_func
        self.q_func = q_func
        self.mu_space = np.array(sorted(mu_space))
        self.beta = beta

        # nu upper bound
        self.V = lambda_rate + np.max(self.mu_space)

        assert np.any(np.isclose(self.mu_space, 0.0))

        self._c_vec = np.array([c_func(i) for i in range(max_state + 1)])
        self._q_vec = np.array([q_func(mu) for mu in self.mu_space])

    def _bellman_rhs(self, i: int, mu: float, J: np.ndarray) -> float:
        # calculate the RHS of Bellman equation
        if i == 0:
            # constraint at 0
            return self._c_vec[i] + (self.V - self.lambda_rate) * J[i] + self.lambda_rate * J[i+1]
        elif i == self.max_state:
            # constraint at M
            return self._c_vec[i] + self.q_func(mu) + mu * J[i-1] + (self.V - mu) * J[i]
        else:
            # other cases
            return (self._c_vec[i] + self.q_func(mu) +
                    mu * J[i-1] + (self.V - self.lambda_rate - mu) * J[i] +
                    self.lambda_rate * J[i+1])

    def value_iteration(self, tolerance: float = 1e-6, max_iter: int = 20000) -> tuple:
        J = np.zeros(self.max_state + 1)
        policy = np.zeros(self.max_state + 1)

        pb = tqdm(range(max_iter))
        for iteration in pb:
            J_new = np.zeros_like(J)

            for i in range(self.max_state + 1):
                # iterate w.r.t. Bellman equation definition
                if i == 0:
                    J_new[i] = self._bellman_rhs(i, 0.0, J) / (self.beta + self.V)
                    policy[i] = 0.0
                else:
                    rhs_values = np.array([
                        self._bellman_rhs(i, mu, J) for mu in self.mu_space
                    ])
                    best_idx = np.argmin(rhs_values)
                    policy[i] = self.mu_space[best_idx]
                    J_new[i] = rhs_values[best_idx] / (self.beta + self.V)

            # check convergence
            diff = np.max(np.abs(J_new - J))
            if diff < tolerance:
                print(f"Value iteration converges at {iteration}th iteration with J difference {diff:.2e}")
                break

            pb.set_postfix({"Diff": f"{diff:.3e}"})
            J = J_new

            if iteration == max_iter - 1:
                print("Reached maximum iterations")

        return J, policy

    def policy_iteration(self, max_iter: int = 500) -> tuple:
        num_states = self.max_state + 1
        policy = np.full(num_states, self.mu_space[0])
        J = np.zeros(num_states)

        for iteration in range(max_iter):
            # setting up linear system AJ = b
            main_diag = np.full(num_states, self.beta + self.V)
            upper_diag = np.full(num_states - 1, -self.lambda_rate)
            lower_diag = np.full(num_states - 1, 0.0)
            b = np.zeros(num_states)

            # case 0
            b[0] = self._c_vec[0]
            main_diag[0] = self.beta + self.lambda_rate
            # upper_diag[0] = -self.lambda_rate 已设置

            # case 1..M-1
            for i in range(1, self.max_state):
                mu_i = policy[i]
                lower_diag[i-1] = -mu_i
                # (β+V)Ji - μJi-1 - (V-λ-μ)Ji - λJi+1 = ci + q(μ)
                # => -(μ)Ji-1 + (β+λ+μ)Ji - λJi+1 = ci + q(μ)
                main_diag[i] = self.beta + self.lambda_rate + mu_i
                b[i] = self._c_vec[i] + self.q_func(mu_i)

            # case M
            mu_M = policy[self.max_state]
            lower_diag[self.max_state - 1] = -mu_M
            main_diag[self.max_state] = self.beta + mu_M
            b[self.max_state] = self._c_vec[self.max_state] + self.q_func(mu_M)

            # solve for AJ = b
            A = diags([lower_diag, main_diag, upper_diag], [-1, 0, 1],
            format='csc')
            J_new = spsolve(A, b)

            # evaluate bellman equation
            if iteration == 0:
                for i in range(min(3, num_states)):
                    if i == 0:
                        lhs = (self.beta + self.lambda_rate) * J_new[i] -
                        self.lambda_rate * J_new[i+1]
                        rhs = b[i]
                    elif i == self.max_state:
```

```python
                        mu = policy[i]
                        lhs = -mu * J_new[i-1] + (self.beta + mu) * J_new[i]
                        rhs = b[i]
                    else:
                        mu = policy[i]
                        lhs = -mu * J_new[i-1] + (self.beta + self.lambda_rate + mu) * J_new[i] - self.lambda_rate * J_new[i+1]
                        rhs = b[i]

            # policy improvement
            new_policy = np.zeros(num_states)
            new_policy[0] = 0.0

            for i in range(1, num_states):
                if i == self.max_state:
                    # edge case
                    rhs_values = np.array([
                        self._c_vec[i] + self.q_func(mu) + mu * J_new[i-1] + (self.V - mu) * J_new[i]
                        for mu in self.mu_space
                    ])
                else:
                    rhs_values = np.array([
                        self._c_vec[i] + self.q_func(mu) + mu * J_new[i-1] +
                        (self.V - self.lambda_rate - mu) * J_new[i] + self.lambda_rate * J_new[i+1]
                        for mu in self.mu_space
                    ])
                best_idx = np.argmin(rhs_values)
                new_policy[i] = self.mu_space[best_idx]

            # convergence check
            policy_diff = np.max(np.abs(new_policy - policy))
            J_diff = np.max(np.abs(J_new - J))

            if policy_diff < 1e-8 and J_diff < 1e-8:
                residual = self.compute_bellman_residual(J_new, new_policy)
                break

            policy = new_policy
            J = J_new.copy()

        return J, policy

    def compute_bellman_residual(self, J: np.ndarray, policy: np.ndarray) -> float:
        """Calculate Bellman Residual"""
        residual = 0.0

        for i in tqdm(range(self.max_state + 1), desc="Evaluating Bellman residual"):
            rhs = self._bellman_rhs(i, policy[i], J)
            T_J = rhs / (self.beta + self.V)
            residual = max(residual, abs(J[i] - T_J))

        return residual

    def compare_policies(self, J_vi: np.ndarray, policy_vi: np.ndarray,
                         J_pi: np.ndarray, policy_pi: np.ndarray,
                         num_states: int = 20):

        print(f"J diff: {np.max(np.abs(J_vi - J_pi)):.10f}")
        print(f"Policy diff: {np.max(np.abs(policy_vi - policy_pi)):.10f}")

        print("\n  State  |  J_VI(i)  |  J_PI(i)  |  ΔJ_VI |  ΔJ_PI | μ*_VI | μ*_PI")
        print("-"*80)

        mismatch_count = 0
        for i in range(min(num_states, self.max_state) + 1):
            delta_J_vi = J_vi[i] - J_vi[i-1] if i > 0 else 0.0
            delta_J_pi = J_pi[i] - J_pi[i-1] if i > 0 else 0.0

            print(f"  {i:3d}   |  {J_vi[i]:8.4f}  |  {J_pi[i]:8.4f}  |  {delta_J_vi:6.2f} |  "
                  f"{delta_J_pi:6.2f} |  {policy_vi[i]:4.1f}  |  {policy_pi[i]:4.1f}  |")

        if mismatch_count == 0:
            print("Test Passed")
        else:
            print(f"Testing Failed with {mismatch_count} different states out of {num_states}")

        residual_vi = self.compute_bellman_residual(J_vi, policy_vi)
        residual_pi = self.compute_bellman_residual(J_pi, policy_pi)
        print(f"\nBellman Residual VI: {residual_vi:.2e}, PI: {residual_pi:.2e}")

def compute_average_cost_steady_state(solver, policy):
    lambda_rate = solver.lambda_rate
    max_state = solver.max_state

    # construct Q matrix
    Q = np.zeros((max_state+1, max_state+1))

    for i in range(max_state+1):
        mu = policy[i]

        if i < max_state:
            Q[i, i+1] = lambda_rate  # arrival

        if i > 0:
            Q[i, i-1] = mu  # service

        # departure
        Q[i, i] = -(lambda_rate + mu)

    # solve πQ = 0, Σπ = 1
    eigenvals, eigenvecs = np.linalg.eig(Q.T)
    zero_idx = np.argmin(np.abs(eigenvals))
    pi = np.abs(eigenvecs[:, zero_idx].real)
    pi /= np.sum(pi)

    # average cost
    costs = np.array([
        solver.c_func(i) + solver.q_func(policy[i])
        for i in range(max_state+1)
    ])
    average_cost = np.dot(pi, costs)
    return average_cost

if __name__ == "__main__":
    # LAMBDA = 10.0
    # MAX_STATE = 100
    # BETA = 0.01

    def linear(x):
        return x

    def quadratic(x):
        return 0.5 * x ** 2

    def exponential(x):
        return np.exp(0.1 * x)


    mu_space = np.linspace(0, 50, 100)

    ls = []

    # for c_func in [linear, quadratic, exponential]:
    #     for q_func in [linear, quadratic, exponential]:

    #         solver = CTMDPControlledQueue(
    #             lambda_rate=LAMBDA,
    #             max_state=MAX_STATE,
    #             c_func=c_func,
    #             q_func=q_func,
    #             mu_space=mu_space,
    #             beta=BETA
    #         )

    #         J_pi, policy_pi = solver.policy_iteration()
    #         ls.append((c_func.__name__, q_func.__name__, J_pi, policy_pi))

    #         from main import ControlledMM1Queue

    #         def service_rate_policy(a):
    #             return policy_pi[a]

    #         queue = ControlledMM1Queue(
    #             arrival_rate=LAMBDA,
    #             service_rate_policy=service_rate_policy,
    #             service_cost_func=q_func,
    #             queue_cost_func=c_func,
    #             max_customers=MAX_STATE
    #         )

    #         queue.run(1e5)
    #         avg_cost = compute_average_cost_steady_state(solver, policy_pi)
    #         print(f"Stationary average cost (steady state): {avg_cost:.4f}")
    #         print(f"Stationary average cost (simulation): 
{queue.get_average_cost():.4f}")

    # plt.figure(figsize=(12, 8), dpi=300)

    # for i, (c_name, q_name, J_pi, policy_pi) in enumerate(ls):
    #     plt.subplot(3, 3, i+1)
    #     plt.plot(policy_pi)
    #     plt.title(f"Queue cost {c_name} + Service cost {q_name}")
    #     plt.grid()

    #     if i > 5:
    #         plt.xlabel("State $i$")

    #     if i % 3 == 0:
    #         plt.ylabel("Optimal Service Speed $\mu_i$")

    # plt.tight_layout()
    # plt.savefig("ctmdp_value_iteration.png")

    c_func = linear
    q_func = quadratic

    for N in [100, 1000, 2000]:
        for beta in [0.001, 0.01, 0.1]:
            for LAMBDA in [5, 10, 20]:
                solver = CTMDPControlledQueue(
                    lambda_rate=LAMBDA,
                    max_state=N,
                    c_func=c_func,
                    q_func=q_func,
                    mu_space=mu_space,
                    beta=beta
                )

                J_pi, policy_pi = solver.policy_iteration()
                ls.append((N, beta, LAMBDA, J_pi, policy_pi))

    fig, axs = plt.subplots(3, 3, figsize=(12, 9), dpi=300)
    axs = axs.flatten()

    for i, (N, beta, LAMBDA, J_pi, policy_pi) in enumerate(ls):
        print(i // 3, N, beta, LAMBDA, policy_pi.shape)
        axs[i//3].plot(policy_pi, label=f"$\lambda = {LAMBDA}$")
        axs[i//3].set_title(f"N={N}, beta={beta}")
        axs[i//3].grid()
        axs[i//3].legend()

    fig.tight_layout()
    fig.savefig("ctmdp_policy_iteration_1.png")

    # print("\n" + "="*80)
    # print("Policy iteration results")
    # print("="*80)
    # solver.compare_policies(J_vi, policy_vi, J_pi, policy_pi)

    # #
    # # np.savez('ctmdp_converged_results.npz',
    # #          J_vi=J_vi, policy_vi=policy_vi,
    # #          J_pi=J_pi, policy_pi=policy_pi,
    # #          lambda_rate=LAMBDA, beta=BETA, max_state=MAX_STATE)

    # avg_cost = compute_average_cost_steady_state(solver, policy_pi)
    # print(f"Stationary average cost: {avg_cost:.4f}")


```