Assignment 2 - Service Control

Ritsaart Bergsma, Jeroen Landman & Witek ten Hove

Assignment 2

Service Rate Control

Consider a discrete-time single-server queueing system that is observed every $\eta > 0$ units of time. The controller makes decisions at times $0, \eta, 2\eta, \ldots$. Jobs arrive following a Poisson distribution with rate 1.5 jobs per period of length η. There is a finite system capacity of eight units; that is, if arriving jobs cause the system content to exceed eight units, excess jobs do not enter the system and are lost.

At each decision epoch, the controller observes the number of jobs in the system and selects the service rate from a set of probability distributions indexed by elements of the set $B = \{0, 1, 2\}$. For each $b \in B$, let $f_b(n)$ denote the probability of n service completions within a period of length η with:

- $f_0(1) = 0.8$, $f_0(2) = 0.2$
- $f_1(1) = 0.5$, $f_1(2) = 0.5$
- $f_2(1) = 0.2$, $f_2(2) = 0.8$

The stationary reward structure consists of four components:

1. A constant reward $R = 5$ for every completed service.
2. An expected holding cost $h(s) = 2s$ per period when there are s jobs in the system.
3. A fixed cost $K = 3$ for changing the service rate.
4. A per-period cost $d(b)$ for using service rate b, where $d(0) = 0$, $d(1) = 2$, and $d(2) = 5$.

Determine a minimum-cost service rate control policy.

(a)

- Formulate the problem above as an infinite horizon Markov decision problem.
- Choose the optimality criterion that you find most reasonable (average costs or discounted costs). Also, choose a method (or methods) for computing the optimal policies and the value. Motivate your choices.
- Develop the model and the algorithm. Compute the optimal policies and the value. *(Note: you should write your own code for the algorithm, i.e., do not use an existing MDP implementation that is available as a code library or on the internet).*

Please Report:

- Model description
- Your choice for an optimality criterion including motivation
- Solution algorithm (including motivation)
- Numerical results and a discussion of those

(b)

Now, we require that the server may work at service rate $b = 2$ at most 25% of the time. Model and solve this adjusted problem.

Solution

Model Description

This model is also described in Puterman's book (section 3.7.2) and we follow the same approach but will motivate the modeling choices.

The transition from state s to s' of the queuing system is as follows:

- After state s the controller chooses a service rate b'.
- After 0, 1, or 2 jobs have departed the system,
- New jobs arrive.
- The current number of jobs in the system is state s'.

Figure 1: System state update between observations t and $t+1$

To formulate this problem as an infinite horizon Markov decision process (MDP), we define the key components:

Decision epochs (T)

$T = \{0, 1, 2, \dots\}$, corresponding to observation times $0, \eta, 2\eta, \dots$

State space (S)

The state $s = (x, b)$ consists of the number of jobs $x \in X$ in the system: $X = \{0, 1, 2, \dots, 8\}$ and the current service rate $b \in B$: $B = \{0, 1, 2\}$.

Action space (A)

At each decision epoch, the controller can take an action $a \in A = \{0, 1, 2\}$ equivalent to selecting a service rate $b' \in B$ for the next epoch.

Transition probabilities (P)

The transition probabilities depend on the arrival rate (Poisson distribution with a rate of 1.5 jobs per period of length η) and the service rate b' chosen.

- Let $f^a(k)$ denote the probability of k job arrivals in a period of length η. Since arrivals follow a Poisson distribution with rate 1.5:

$$f_a(k) = \begin{cases} \frac{(1.5)^k e^{-1.5}}{k!}, & \text{if } k = 0, 1, 2, \dots \\ 0, & \text{otherwise} \end{cases}$$

- Let $f_b(n)$ denote the probability of $n \in \{1, 2\}$ service completions under service rate b, where the distributions of n are given for each $b \in B$. This only applies to starting states s with $x > 1$ when the system contains enough jobs to be processed.

Define $p(s'|s, a)$ as the probability of transitioning to state $s' = (x', b')$ when the system is in state $s = (x, b)$ and action $a = b'$ is chosen. We can then distinguish the following cases:

$$p(s'|s, a \neq b') = 0$$

$$p(s'|s, b') = \begin{cases} f^a(x') & \text{if } x \in \{0, 1\} \text{ and } x' < 8, \\ 1 - \sum_{k=0}^{7} f^a(k) & \text{if } x \in \{0, 1\} \text{ and } x' = 8, \\ \sum_{n=1}^{2} f_{b'}(n) \cdot f^a(x' - x + n) & \text{if } 1 < x \leq 8 \text{ and } x' < 8, \\ \sum_{n=1}^{2} f_{b'}(n) \cdot \left(1 - \sum_{k=0}^{7-x+n} f^a(k)\right) & \text{if } 1 < x \leq 8 \text{ and } x' = 8. \end{cases}$$

Cost and rewards (C)

The cost function consists of four components:

1. **Expected reward for service completions**: For each random number of service completions $n \sim f_b(n)$ the system receives a reward of $R = 5$. The expected total reward for service completions when the system is in state s and action $a = b'$ is chosen equals: $5 \cdot \mathbb{E}(n)$.
2. **Holding cost**: The expected holding cost per period is $h(x) = 2x$ when there are x jobs in the system.
3. **Cost of changing service rate**: If the controller changes the service rate from one period to the next ($b \neq b'$), a fixed cost of $K = 3$ is incurred.
4. **Per-period cost for using service rate**: The cost $d(b')$ depends on the chosen service rate b', where: $d(0) = 0, \quad d(1) = 2, \quad d(2) = 5$

Define the cost function $C(s, a)$ for state $s = (x, b)$ and chosen action (service rate) a as follows:

$$C(s,a) = \begin{cases} d(a) + K \cdot \mathbb{1}_{\{b \neq a\}}, & \text{if } s = 0 \\ -R + h(1,H) + d(a) + K \cdot \mathbb{1}_{\{b \neq a\}}, & \text{if } s = 1 \\ -R \cdot \sum_{n=1}^{2} f_a(n) \cdot n + h(x,H) + d(a) + K \cdot \mathbb{1}_{\{b \neq a\}}, & \text{if } s > 1 \end{cases}$$

where:

- $d(a)$ is the per-period cost of using service rate a.
- $K \cdot \mathbb{1}_{\{b \neq a\}}$ represents the cost K for changing the service rate (incurred only if $b \neq a$).
- R is the reward for processing a job.
- $h(x, H)$ is the holding cost, where H is a constant per-unit cost and x is the number of jobs in the system in state s.
- $f_a(n)$ is the probability of processing n jobs under chosen service rate $b = a$.

Optimality criterion

The goal is to find the optimal policy that minimizes the expected discounted cost or average cost per period. The optimal policy is a stationary policy that prescribes the service rate to choose at each state.

Assumption 1

For each stationary policy π, the associated Markov chain $\{X_n\}$ has a single recurrent class of states (Unichain Assumption).
Proof:

1. Choose arbitrary $s(x+1,b')$. There exists a path from $s(x+1,b')$ to $s(x,b)$ with $P(k=n-1)>0$ under policy $\pi(s(x+1,b'))$. This is evident as k can take any value from 0 to ∞ with probability larger than 0 and $n \subset k$ with positive probabilities for all n.
2. Choose arbitrary $s(x-1,b')$. There exists a path from $s(x-1,b')$ to $s(x,b)$ with $P(k=n+1)>0$ under policy $\pi(s(x-1,b'))$. Same as above.
3. Choose arbitrary $s(x,b)$. There exists a one-step path from $s(x,b)$ to itself with $P(k=x)>0$ if $x=\{0,1\}$ and $P(k=n)>0$ if $x>1$ under policy $\pi(s(x,b))$. Same as above.

By recurrence and transitivity, all states under policy π are in the same recurrent class. Therefore, the Markov chain is unichain.

For solving an infinite MDP problem there are several methods available. The most common methods are:

1. **Value Iteration**: This method iteratively computes the value function for each state by updating the value based on the Bellman equation. The policy is then derived from the value function.
2. **Policy Iteration**: This method alternates between policy evaluation and policy improvement steps. It evaluates the current policy and then improves it by selecting the best action at each state.
3. **Linear Programming**: The MDP problem can be formulated as a linear program, and the optimal policy can be obtained by solving the linear program.

For this particular instance we will use the linear programming approach to solve the MDP problem. Although part a of the problem set could be solved with all of the above mentioned methods, the linear programming approach is particularly well suited for part b problem as it allows for a direct formulation of the MDP problem and the addition of special constraints, in this case the constraint that the server may work at service rate $b=2$ at most 25% of the time.

The Linear Programming Algorithm

- **Define** $q_{s,a}$: the long-run fraction of decision epochs at which the system is in state s and action a is chosen.

- So, $\sum_{a\in A(s)} q_{s,a}$: the long-run fraction of decision epochs at which the system is in state s.
- Also, $\sum_{s\in S}\sum_{a\in A(s)} q_{s,a} = 1$.
- **Under the unichain assumption**, an average cost optimal policy can be computed using:

  Minimize

$$\sum_{s\in S}\sum_{a\in A(s)} C(s,a)q_{s,a}$$

  Subject to

$$\sum_{s\in S}\sum_{a\in A(s)} p(s'|s,a)q_{s,a} = \sum_{a\in A(s)} q_{s',a}, \quad \forall s' \in S$$

$$\sum_{s\in S}\sum_{a\in A(s)} q_{s,a} = 1$$

$$q_{s,a} \geq 0, \quad \forall a \in A(s), s \in S$$

Code

Setup

Parameters:

- **State Space (S)**: Each state is a tuple (x, b), where:
  - x is the number of jobs in the system (ranging from 0 to $X_{\max} = 8$).
  - b is the current service rate (from the set $B = \{0, 1, 2\}$).
- **Action Space (A)**: The actions are the possible service rates $b' \in B$ that the controller can choose at each decision epoch.
- **Transition Probabilities (P)**: The function `P(x_i, b_i, x_j, b_j, a)` calculates the probability of moving from state (x_i, b_i) to (x_j, b_j) given action a. It accounts for:
  - The probability of job arrivals (modeled using a Poisson distribution with rate 1.5).
  - The probability of job completions based on the chosen service rate's distribution $f_b(n)$.

- **Reward Function** ($r(x, b, a)$): Computes the immediate reward for being in state $s = (x, b)$ and choosing action a. It includes:
  - **Processing Rewards**: Earned for completing jobs.
  - **Holding Costs**: Penalty for the number of jobs in the system.
  - **Operating Costs**: Cost of using a specific service rate.
  - **Switching Costs**: Additional cost if the service rate changes (i.e., $b \neq a$).

```python
##Initialize packages, parameters and functions

from ortools.linear_solver import pywraplp
from scipy.stats import poisson

# Parameters
d = { # cost for using server rate
        0: 0,
        1: 2,
        2: 5
    }
K = 3  # cost for changing the service rate
R = 5  # reward for processing 1 job per period
H = 2 # holding cost per period
X_max = 8  # max system capacity
l = 0.95 # discount factor lambda

# cost function for calculating the holding cost
def h(s, H):
    return H * s # expected holding cost per period

# probability f_b(n) of processing n jobs if the service rate is b
def f(n, b):
    b_prob = {0: [0.8, 0.2], 1: [0.5, 0.5], 2: [0.2, 0.8]}
    return b_prob[b][n - 1]

# probability g(n) of n new jobs arriving
def g(n):
    return poisson.pmf(n, mu=1.5)

# probability of n or more new jobs arriving
def g_or_more(n):
    return 1 - sum(g(m) for m in range(n))

# Sets of states (S: number of jobs in the system, B: server states)
```

```
X = range(X_max + 1)
B = [0, 1, 2]

# Transition probabilities
def P(x_i, b_i, x_j, b_j, a):
    if b_j != a:
        return 0.0
    if x_i <= 1:
        if x_j < X_max:
            return g(x_j) # Arrival rate
        elif x_j == X_max:
            return g_or_more(x_j) # 1 - probability of 0, 1, 2, 3, 4, 5, 6, 7 arrivals
        else:
            return 0.0 # x_j > X_max, not possible
    elif x_i <= X_max:
        if x_j < X_max:
            return sum(f(n, b_j) * g(x_j - x_i + n) for n in [1, 2]) # Arrivals have to compe
        elif x_j == X_max:
            return sum(f(n, b_j) * g_or_more(x_j - x_i + n) for n in [1, 2]) # Arrivals have
        else:
            return 0.0 # x_j > X_max, not possible
    else:
        return 0.0 # x_i > X_max, not possible

# Define the expected reward R_{(s,b),a'} when state is (s,b) and chosen service rate is a'
def r(x, b, a):
    if x == 0:
        return d[a] + K*(b != a) # no costs for processing and holding jobs
    elif x == 1:
        return  - R + h(1, H) + d[a] + K*(b != a) # reward and cost for processing and holdi
    else:
        return - R * sum(f(n, a) * n for n in [1, 2]) + h(x, H) + d[a] + K*(b != a)
```

Optimality criterion: discounted cost

In this solution, we model the problem as a Markov Decision Process (MDP) with the goal of **minimizing the expected discounted cost** over an infinite horizon. The discount factor $\lambda = 0.95$ is applied to future costs, reflecting the preference for distant costs over immediate ones.

Code Explanation:

- **Initialization**:

  - Import necessary libraries (`pywraplp` for the solver and `poisson` from `scipy.stats`).
  - Define parameters such as costs, rewards, and probabilities.
  - Define helper functions for holding costs (`h`), processing probabilities (`f`), and arrival probabilities (`g` and `g_or_more`).

- **Decision Variables**:
  - Create variables $q_{(x,b),a}$ representing the probability of being in state (x, b) and choosing action a.
  - These variables are created using `solver.NumVar`.

- **Objective Function**:
  - Constructed to minimize the expected discounted costs
  - Uses the immediate reward cost `r(x, b, a)` and sums over all states and actions.
  - Set up using `solver.Objective` and `objective.SetCoefficient`.

- **Balance Constraints**:
  - Ensure the flow of probabilities between states adheres to the MDP dynamics.
  - Incorporate the discount factor λ.
  - Added using `solver.Add`, ensuring that the probability of being in a state equals the discounted sum of probabilities of transitioning into that state.

- **Solving and Output**:
  - The solver is executed with `solver.Solve()`.
  - Upon finding an optimal solution, the code prints the optimal actions for each state where the decision variable x is greater than zero.

```
# Create the solver
solver = pywraplp.Solver.CreateSolver('GLOP')

# Decision variables q_{(x,b),b'}: probability of choosing server state b' when state is (x,
q = {}
for x_i in X:
    for b in B:
        for b_prime in B:
            q[(x_i, b, b_prime)] = solver.NumVar(0.0, 1.0, f'q_{x_i}_{b}_{b_prime}')

# Initial state probabilities alpha
alpha = {}
n_total_states = len(X) * len(B)
for x_i in X:
    for b in B:
```

```python
        alpha[(x_i, b)] = 1 / n_total_states

# Objective: maximize the expected reward
objective = solver.Objective()
for x_i in X:
    for b in B:
        for a in B:
            objective.SetCoefficient(q[(x_i, b, a)], r(x_i, b, a))
objective.SetMinimization()

# Balance constraints
for x_j in X:
    for b_j in B:
        lhs = solver.Sum(q[(x_j, b_j, a)] for a in B)
        rhs = l * solver.Sum(
            P(x_i, b_i, x_j, b_j, a) * q[(x_i, b_i, a)]
            for x_i in X for b_i in B for a in B
        )
        solver.Add(lhs == alpha[(x_j, b_j)] + rhs)

# Solve the problem
status = solver.Solve()

# Print the optimal solution
if status == pywraplp.Solver.OPTIMAL:
    for x_i in X:
        print(f"If {x_i} jobs in the system:")
        for b in B:
            print(f"   If previous service rate was {b}:")
            for a in B:
                x_opt = q[(x_i, b, a)].solution_value()
                if x_opt > 0:
                    print(f"      Use service rate {a} (x={x_opt})")
else:
    print("No feasible solution found.")
```

```
If 0 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (x=0.5515694772088197)
   If previous service rate was 1:
      Use service rate 1 (x=0.412440442871158)
   If previous service rate was 2:
      Use service rate 0 (x=0.18750922188314792)
If 1 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (x=1.0)
      Use service rate 1 (x=0.009833129370600473)
   If previous service rate was 1:
      Use service rate 1 (x=0.8271892020120805)
   If previous service rate was 2:
      Use service rate 0 (x=0.4882627887525087)
If 2 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (x=1.0)
      Use service rate 1 (x=0.04305622179645904)
   If previous service rate was 1:
      Use service rate 1 (x=1.0)
      Use service rate 2 (x=0.041286720618086234)
   If previous service rate was 2:
      Use service rate 2 (x=0.8460416296644189)
If 3 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (x=0.7410947078975391)
   If previous service rate was 1:
      Use service rate 1 (x=1.0)
      Use service rate 2 (x=0.10803528137180927)
   If previous service rate was 2:
      Use service rate 1 (x=0.09916337890020244)
      Use service rate 2 (x=1.0)
If 4 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (x=0.400015575812334)
   If previous service rate was 1:
      Use service rate 1 (x=1.0)
      Use service rate 2 (x=0.10059731508881485)
   If previous service rate was 2:
      Use service rate 1 (x=0.17800895119033391)
      Use service rate 2 (x=1.0)
If 5 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (x=0.1826868371621025)
   If previous service rate was 1:
      Use service rate 1 (x=1.0)
      Use service rate 2 (x=0.04382019884480864)
   If previous service rate was 2:
      Use service rate 1 (x=0.10888365390752497)
      Use service rate 2 (x=1.0)
If 6 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (x=0.08440583789934589)
   If previous service rate was 1:
      Use service rate 1 (x=0.9816579436923619)
   If previous service rate was 2:
      Use service rate 2 (x=0.9996963146483523)
If 7 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (x=0.049913989087310186)
   If previous service rate was 1:
      Use service rate 1 (x=0.8519854940975995)
   If previous service rate was 2:
      Use service rate 2 (x=0.7751528353258794)
If 8 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (x=0.04077194305373725)
   If previous service rate was 1:
      Use service rate 1 (x=1.0)
      Use service rate 2 (x=0.05292823168301868)
   If previous service rate was 2:
      Use service rate 2 (x=0.7939926761596431)
```

Optimality criterion: average cost

In this solution, the objective shifts to **minimizing the average cost per period** without considering discounting. This reflects a long-term perspective where future cost are valued equally with immediate cost

Solution Design:

- **Balance Constraints**:
  - Modified to ensure that the steady-state probabilities satisfy the flow conservation equations.
  - The inflow to each state equals the outflow, reflecting a stable long-term system.
- **Total Probability Constraint**:
  - An additional constraint ensures that the sum of all probabilities $q_{(x,b),a}$ equals 1.
  - This ensures that the decision variables represent a valid probability distribution.

Code Explanation:

- **Initialization**: Similar to the previous model.
- **Decision Variables and Objective Function**:
  - Variables $q_{(x,b),a}$ are defined in the same way.
  - The objective function minimizes the average cost, summing over all states and actions without discounting.
- **Balance Constraints**:
  - The constraints now equate the total inflow and outflow probabilities for each state (x_j, b_j).
  - Implemented using `solver.Add`, ensuring that for each state, the sum of incoming probabilities equals the sum of outgoing probabilities.
- **Total Probability Constraint**:
  - Ensures the sum of all $q_{(x,b),a}$ is 1.
  - Added using `solver.Add`.
- **Solving and Output**:
  - The solver is executed, and upon finding an optimal solution, the code prints the optimal actions for each state.

```
# Create the solver
solver = pywraplp.Solver.CreateSolver('GLOP')
solver.SetTimeLimit(60000)  # Set time limit in milliseconds (60 seconds)

# Decision variables q_{(x,b),b'}: probability of choosing server state b' when state is (x,
q = {}
for x_i in X:
    for b in B:
        for b_prime in B:
            q[(x_i, b, b_prime)] = solver.NumVar(0.0, 1.0, f'q_{x_i}_{b}_{b_prime}')

# Objective: minimize the expected cost
objective = solver.Objective()
for x_i in X:
    for b in B:
        for a in B:
            objective.SetCoefficient(q[(x_i, b, a)], r(x_i, b, a))
objective.SetMinimization()

# Balance constraints
```

```
for x_j in X:
    for b_j in B:
        # Outflow: the sum of probabilities of choosing each action a in state (x_j, b_j)
        outflow = solver.Sum(q[(x_j, b_j, a)] for a in B)

        # Inflow: the weighted sum of transition probabilities for each incoming state-actio
        inflow = solver.Sum(
            P(x_i, b_i, x_j, b_j, a) * q[(x_i, b_i, a)]
            for x_i in X for b_i in B for a in B
        )

        # Add the constraint that the inflow and outflow must be equal
        solver.Add(inflow == outflow)

# Probability constraint to ensure all probabilities sum to 1
solver.Add(solver.Sum(q[(x_i, b, a)] for x_i in X for b in B for a in B) == 1)

# Solve the problem
status = solver.Solve()

# Print the optimal solution
if status == pywraplp.Solver.OPTIMAL:
    for x_i in X:
        print(f"If {x_i} jobs in the system:")
        for b in B:
            print(f"   If previous service rate was {b}:")
            for a in B:
                x_opt = q[(x_i, b, a)].solution_value()
                if x_opt > 0:
                    print(f"      Use service rate {a} (x={x_opt})")
else:
    print("No feasible solution found.")
```

```
If 0 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (x=0.0585048113191524444)
   If previous service rate was 1:
   If previous service rate was 2:
      Use service rate 0 (x=0.018712275632014764)
If 1 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (x=0.10380210394354188)
   If previous service rate was 1:
   If previous service rate was 2:
      Use service rate 0 (x=0.06320413392027384)
If 2 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (x=0.0898852431812663)
   If previous service rate was 1:
   If previous service rate was 2:
      Use service rate 2 (x=0.10482825147644233)
If 3 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (x=0.05095945420243811)
   If previous service rate was 1:
   If previous service rate was 2:
      Use service rate 2 (x=0.11966770095213014)
If 4 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (x=0.02136610755534112)
   If previous service rate was 1:
   If previous service rate was 2:
      Use service rate 2 (x=0.110053529840278)
If 5 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (x=0.007086725935430403)
   If previous service rate was 1:
   If previous service rate was 2:
      Use service rate 2 (x=0.08893112299419272)
If 6 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (x=0.001940904901064615)
   If previous service rate was 1:
   If previous service rate was 2:
      Use service rate 2 (x=0.06861018635393497)
If 7 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (x=0.00045217035391534874)
   If previous service rate was 1:
   If previous service rate was 2:
      Use service rate 2 (x=0.04722555557607802)
If 8 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (x=0.00011104660409901706)
   If previous service rate was 1:
```

```
If previous service rate was 2:
   Use service rate 2 (x=0.04465867525840605)
```

Optimality criterion: discounted cost with service rate restriction

This solution introduces a **service rate restriction** into the discounted cost model, limiting the usage of the highest service rate $b = 2$ to at most 25% of the time. This constraint could represent resource limitations or cost considerations.

Solution Design:

- **Service Rate Restriction**:
  - A new constraint is added to limit the total probability of choosing service rate $a = 2$.
  - The constraint ensures that the sum of probabilities where $a = 2$ does not exceed 25% of the total probability.

Code Explanation:

- **Initialization, Decision Variables, and Objective Function**: Same as in the discounted cost model.
- **Balance Constraints**: Remain the same.
- **Service Rate Restriction Constraint**:
  - Calculated the total probability of choosing service rate $a = 2$ across all states.

```python
# Create the solver
solver = pywraplp.Solver.CreateSolver('GLOP')

# Decision variables q_{(x,b),b'}: probability of choosing server state b' when state is (x,
q = {}
for x_i in X:
    for b in B:
        for b_prime in B:
            q[(x_i, b, b_prime)] = solver.NumVar(0.0, 1.0, f'q_{x_i}_{b}_{b_prime}')

# Initial state probabilities alpha
alpha = {}
n_total_states = len(X) * len(B)
for x_i in X:
    for b in B:
        alpha[(x_i, b)] = 1 / n_total_states
```

```
# Objective: minimize the expected cost
objective = solver.Objective()
for x_i in X:
    for b in B:
        for a in B:
            objective.SetCoefficient(q[(x_i, b, a)], r(x_i, b, a))
objective.SetMinimization()

# Balance constraints
for x_j in X:
    for b_j in B:
        lhs = solver.Sum(q[(x_j, b_j, a)] for a in B)
        rhs = l * solver.Sum(
            P(x_i, b_i, x_j, b_j, a) * q[(x_i, b_i, a)]
            for x_i in X for b_i in B for a in B
        )
        solver.Add(lhs == alpha[(x_j, b_j)] + rhs)

# Define the left-hand side: the total probability of using server state b = 2
service_rate_b2 = solver.Sum(q[(x_i, 2, a)] for x_i in X for a in B)

# Define the right-hand side: 25% of the total probability across all states, server states,
total_probability = solver.Sum(q[(x_i, b, a)] for x_i in X for b in B for a in B)
solver.Add(service_rate_b2 <= 0.25 * total_probability)

# Solve the problem
status = solver.Solve()

# Print the optimal solution
if status == pywraplp.Solver.OPTIMAL:
    for x_i in X:
        print(f"If {x_i} jobs in the system:")
        for b in B:
            print(f"   If previous service rate was {b}:")
            for a in B:
                q_opt = q[(x_i, b, a)].solution_value()
                if q_opt > 0:
                    print(f"      Use service rate {a} (q={q_opt})")
else:
    print("No feasible solution found.")
```

```
If 0 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (q=0.5391421528072634)
   If previous service rate was 1:
      Use service rate 0 (q=0.34956839328894035)
   If previous service rate was 2:
      Use service rate 0 (q=0.043287273129876985)
If 1 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (q=1.0)
   If previous service rate was 1:
      Use service rate 0 (q=0.05381167876673675)
      Use service rate 1 (q=0.7182834160071975)
   If previous service rate was 2:
      Use service rate 0 (q=0.18290524885797413)
If 2 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (q=1.0)
      Use service rate 1 (q=0.10515373112308939)
   If previous service rate was 1:
      Use service rate 1 (q=1.0)
   If previous service rate was 2:
      Use service rate 1 (q=0.4070549388669085)
      Use service rate 2 (q=0.03685738787409642)
If 3 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (q=0.9488530104689104)
   If previous service rate was 1:
      Use service rate 1 (q=1.0)
      Use service rate 2 (q=0.042313706560176176)
   If previous service rate was 2:
      Use service rate 2 (q=0.7533647733807503)
If 4 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (q=0.651850695233119)
   If previous service rate was 1:
      Use service rate 1 (q=1.0)
      Use service rate 2 (q=0.03477756618579748)
   If previous service rate was 2:
      Use service rate 2 (q=0.9168260369181563)
If 5 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (q=0.4392819054195445)
   If previous service rate was 1:
      Use service rate 1 (q=1.0)
      Use service rate 2 (q=0.02338929911117748)
   If previous service rate was 2:
      Use service rate 2 (q=0.8715618579909199)
If 6 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (q=0.30586608182725467)
      Use service rate 1 (q=0.12869176334882293)
   If previous service rate was 1:
      Use service rate 1 (q=1.0)
      Use service rate 2 (q=0.005765685332450447)
   If previous service rate was 2:
      Use service rate 2 (q=0.7356895853173471)
If 7 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (q=0.5775347744027353)
   If previous service rate was 1:
      Use service rate 1 (q=0.8931160122834021)
   If previous service rate was 2:
      Use service rate 2 (q=0.5353403005392913)
If 8 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (q=1.0)
      Use service rate 1 (q=0.06697829922460165)
   If previous service rate was 1:
      Use service rate 0 (q=0.1156218286087753)
      Use service rate 1 (q=1.0)
   If previous service rate was 2:
      Use service rate 2 (q=0.517112597124683)
```

Optimality criterion: average cost with service rate restriction

This solution applies the **service rate restriction** to the average cost model, limiting the usage of the highest service rate $b = 2$ while aiming to minimize the average cost per period.

Solution Design:

- **State and Action Spaces, Transition Probabilities, and Reward Function**: Same as in the average cost model.
- **Service Rate Restriction**:

  – Similar to the previous solution, a constraint is added to limit the total probability of choosing service rate $a = 2$.

Code Explanation:

- **Initialization, Decision Variables, Objective Function, and Balance Constraints**: Same as in the average cost model.
- **Total Probability Constraint**: Ensures the probabilities sum to 1.

```
# Create the solver
solver = pywraplp.Solver.CreateSolver('GLOP')
solver.SetTimeLimit(60000)  # Set time limit in milliseconds (60 seconds)

# Decision variables q_{(x,b),b'}: probability of choosing server state b' when state is (x,
q = {}
for x_i in X:
    for b in B:
        for b_prime in B:
            q[(x_i, b, b_prime)] = solver.NumVar(0.0, 1.0, f'q_{x_i}_{b}_{b_prime}')

# Objective: minimize the expected cost
objective = solver.Objective()
for x_i in X:
    for b in B:
        for a in B:
            objective.SetCoefficient(q[(x_i, b, a)], r(x_i, b, a))
objective.SetMinimization()

# Balance constraints
for x_j in X:
    for b_j in B:
        # Outflow: the sum of probabilities of choosing each action a in state (x_j, b_j)
        outflow = solver.Sum(q[(x_j, b_j, a)] for a in B)

        # Inflow: the weighted sum of transition probabilities for each incoming state-actio
        inflow = solver.Sum(
            P(x_i, b_i, x_j, b_j, a) * q[(x_i, b_i, a)]
            for x_i in X for b_i in B for a in B
        )

        # Add the constraint that the inflow and outflow must be equal
        solver.Add(inflow == outflow)
```

```
# Probability constraint to ensure all probabilities sum to 1
solver.Add(solver.Sum(q[(x_i, b, a)] for x_i in X for b in B for a in B) == 1)

# Service rate restriction: Limit the probability of using server state b = 2 to 25%
service_rate_limit = solver.Sum(q[(x_i, 2, a)] for x_i in X for a in B)
solver.Add(service_rate_limit <= 0.25)

# Solve the problem
status = solver.Solve()

# Print the optimal solution
if status == pywraplp.Solver.OPTIMAL:
    for x_i in X:
        print(f"If {x_i} jobs in the system:")
        for b in B:
            print(f"   If previous service rate was {b}:")
            for a in B:
                q_opt = q[(x_i, b, a)].solution_value()
                if q_opt > 0:
                    print(f"      Use service rate {a} (q={q_opt:.2f})")
else:
    print("No feasible solution found.")
```

```
If 0 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (q=0.00)
   If previous service rate was 1:
      Use service rate 1 (q=0.05)
   If previous service rate was 2:
If 1 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (q=0.01)
   If previous service rate was 1:
      Use service rate 1 (q=0.10)
   If previous service rate was 2:
      Use service rate 0 (q=0.01)
If 2 jobs in the system:
   If previous service rate was 0:
      Use service rate 0 (q=0.01)
   If previous service rate was 1:
      Use service rate 1 (q=0.12)
   If previous service rate was 2:
      Use service rate 1 (q=0.02)
If 3 jobs in the system:
   If previous service rate was 0:
      Use service rate 1 (q=0.00)
   If previous service rate was 1:
      Use service rate 1 (q=0.11)
   If previous service rate was 2:
      Use service rate 2 (q=0.03)
If 4 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (q=0.00)
   If previous service rate was 1:
      Use service rate 1 (q=0.09)
   If previous service rate was 2:
      Use service rate 2 (q=0.04)
If 5 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (q=0.00)
   If previous service rate was 1:
      Use service rate 1 (q=0.07)
   If previous service rate was 2:
      Use service rate 2 (q=0.05)
If 6 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (q=0.00)
   If previous service rate was 1:
      Use service rate 1 (q=0.04)
      Use service rate 2 (q=0.02)
   If previous service rate was 2:
      Use service rate 2 (q=0.04)
If 7 jobs in the system:
   If previous service rate was 0:
      Use service rate 2 (q=0.00)
   If previous service rate was 1:
      Use service rate 1 (q=0.05)
   If previous service rate was 2:
      Use service rate 2 (q=0.03)
If 8 jobs in the system:
   If previous service rate was 0:
      Use service rate 1 (q=0.00)
   If previous service rate was 1:
      Use service rate 1 (q=0.07)
   If previous service rate was 2:
```

```
Use service rate 2 (q=0.03)
```

General Notes on the Code:

- **Solver Configuration**:
  - All models use the OR-Tools `GLOP` linear solver.
  - Time limits are set where necessary to prevent long computation times.
- **Printing Results**:
  - The code checks if an optimal solution is found.
  - It iterates over all states and prints the optimal action(s) where the decision variable x is positive, indicating the action(s) to take in each state.