# Optimization under Uncertainty: Assignment 3 Report

Hariaksh Pandya - s2692608

April 4, 2025

## Contents

# 1 Q1: Benders Decomposition for a Two-Stage Problem

This question revisits the two-stage stochastic supply chain model from Assignment 2, Question 3, focusing on solving it using Benders decomposition. Two-stage stochastic optimization deals with problems where decisions are made sequentially in the face of uncertainty. The problem involves first-stage decisions ($x_n$: initial shipment quantity to city $n$) made before uncertainty (scenario-specific demands $d_n^k$) is revealed. Second-stage (recourse) decisions ($u_n^k, v_n^k, z_n^k, s_n^k$: adjustments, leftovers, shortages) are made after a scenario $k$ materializes, aiming to minimize costs given the first-stage choice and the realized demand.

The objective is to minimize the sum of the deterministic first-stage costs and the *expected* cost of the second-stage actions. Benders decomposition (L-shaped method) is applied to solve this problem, exploiting its structure by separating the first stage from the scenario-dependent second stages.

## 1.1 Mathematical Formulation Recap

- **Sets & Indices:** $\mathcal{N}$: cities ($n$), $\mathcal{K}$: scenarios ($k$).
- **Parameters:** $I$: depot capacity, $Y_n$: initial city inventory, $\theta_n$: 1st-stage cost, $\theta'_n$: 2nd-stage adjustment cost, $h, g$: leftover/shortage costs, $d_n^k$: demand, $p_k$: scenario probability ($= 1/|\mathcal{K}|$ here).
- **First-Stage Variables:** $x_n \geq 0$: shipment to city $n$.
- **Second-Stage Variables (scenario $k$, given $x$):** $u_n^k, v_n^k, z_n^k, s_n^k \geq 0$.
- **Objective Function:** Minimize $\sum_{n\in\mathcal{N}} \theta_n x_n + \sum_{k\in\mathcal{K}} p_k Q_k(x)$
- **First-Stage Constraints:** $\sum_{n\in\mathcal{N}} x_n \leq I$
- **Second-Stage Problem Definition ($Q_k(x)$):**

$$Q_k(x) = \min \sum_{n\in\mathcal{N}} [\theta'_n(u_n^k + v_n^k) + hz_n^k + gs_n^k]$$

subject to:

$$\begin{aligned}
I + \sum_{n\in\mathcal{N}} u_n^k &\geq \sum_{n\in\mathcal{N}} v_n^k + \sum_{n\in\mathcal{N}} x_n && (\pi_{cap}^k : \text{Capacity Dual}) \\
Y_n + x_n + v_n^k + s_n^k &= d_n^k + z_n^k + u_n^k && \forall n \in \mathcal{N} \quad (\pi_{dem,n}^k : \text{Demand Dual}) \\
u_n^k, v_n^k, z_n^k, s_n^k &\geq 0 && \forall n \in \mathcal{N}
\end{aligned}$$

$Q_k(x)$ is convex and piecewise-linear in $x$.

## 1.2 Benders Decomposition Approaches and Algorithm

Benders decomposition iteratively builds an approximation of the recourse function(s) in a master problem using cuts derived from the dual solutions of the scenario subproblems.

- **Multi-Cut Benders:** Uses variables $\eta_k \approx Q_k(x)$. Master objective: $\min \sum \theta_n x_n + \sum p_k \eta_k$. Adds optimality cuts $\eta_k \geq (\pi_k)^T(h^k - T^k x)$ for violated scenarios.
- **Single-Cut Benders:** Uses variable $\theta \approx \sum p_k Q_k(x)$. Master objective: $\min \sum \theta_n x_n + \theta$. Adds one aggregated cut $\theta \geq \sum p_k (\pi_k)^T(h^k - T^k x)$ if $\hat{\theta}$ is too low.

**Algorithm Outline:**

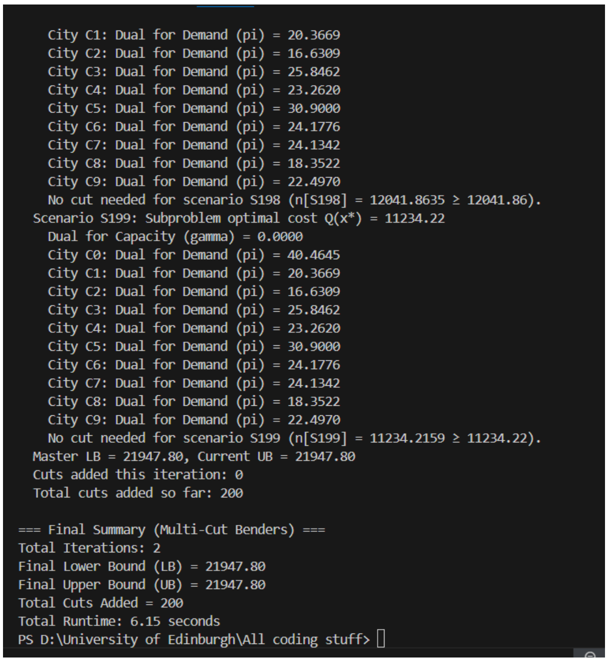
1. **Initialization:** Set $t = 1$, LB = $-\infty$, UB = $+\infty$. Initialize Master Problem (MP) with first-stage constraints and possibly trivial bounds on epigraph variables ($\eta_k \geq 0$ or $\theta \geq 0$).
2. **Master Solve:** Solve MP $\rightarrow (\hat{x}^t, \hat{\eta}^t$ or $\hat{\theta}^t)$, objective $\hat{\nu}_t$. Update LB = $\max(\text{LB}, \hat{\nu}_t)$.
3. **Subproblem Solve:** For $k = 1..K$, solve scenario $k$'s second-stage dual problem (or primal) given $\hat{x}^t$. Get optimal value $Q_k(\hat{x}^t)$ and dual solution $\pi_k = (\pi^k_{cap}, \{\pi^k_{dem,n}\}_n)$ (or dual ray $r_k$ if infeasible).
4. **Check Cut:**
   - If any subproblem $k$ is infeasible: Add feasibility cut $0 \geq (r_k)^T(h^k - T^k x)$ to MP. Set `cut_added = true`. Go to step 6.
   - Calculate potential UB $= \sum \theta_n \hat{x}^t_n + \sum p_k Q_k(\hat{x}^t)$. Update UB $= \min(\text{UB}, \text{potential UB})$.
   - Check for violated optimality cuts (using tolerance $\delta$):
     - Multi-Cut: If $\hat{\eta}^t_k < Q_k(\hat{x}^t) - \delta$, add cut $\eta_k \geq (\pi_k)^T(h^k - T^k x)$. Set `cut_added = true` if any cut added.
     - Single-Cut: If $\hat{\theta}^t < \sum p_k Q_k(\hat{x}^t) - \delta$, add cut $\theta \geq \sum p_k(\pi_k)^T(h^k - T^k x)$. Set `cut_added = true`.
5. **Terminate?** Stop if UB - LB $\leq \epsilon$, or `cut_added` is false, or iteration limit (200) reached.
6. **Iterate:** $t \leftarrow t + 1$. Go to Step 2.

*Flowchart:* (Start) $\rightarrow$ [Solve Master Problem (MP)] $\rightarrow$ Get $(\hat{x}, \hat{\eta}$ or $\hat{\theta})$, LB $\rightarrow$ [Solve Subproblems (SPs) for $\hat{x}$] $\rightarrow$ For each SP $k$, get $Q_k(\hat{x})$, $\pi_k$ or $r_k$. $\rightarrow$ (Check Feasibility) $\rightarrow$ If Infeasible: [Add Feasibility Cut(s)] $\rightarrow$ (To Step 6). $\rightarrow$ If Feasible: Calculate UB $\rightarrow$ (Check Optimality Cuts) $\rightarrow$ If Violated: [Add Optimality Cut(s)] $\rightarrow$ (To Step 6). $\rightarrow$ If Not Violated: (Check Convergence: UB-LB $\leq \epsilon$?) $\rightarrow$ If Yes: [Terminate]. $\rightarrow$ If No: (To Step 6). $\rightarrow$ [Step 6: Iterate t=t+1] $\rightarrow$ (Back to Solve MP).

**Implementation Notes:** A cut violation tolerance $\delta > 0$ is crucial. The implementation should use dual solutions $\pi_k$ to construct cuts. Assuming relatively complete recourse means feasibility cuts are theoretically unnecessary for feasible $x$ from the MP satisfying first-stage constraints. However, a robust implementation often includes checks for subproblem infeasibility to handle potential modeling issues or numerical instability, generating feasibility cuts if needed.[2]

### 1.3 Results and limitations

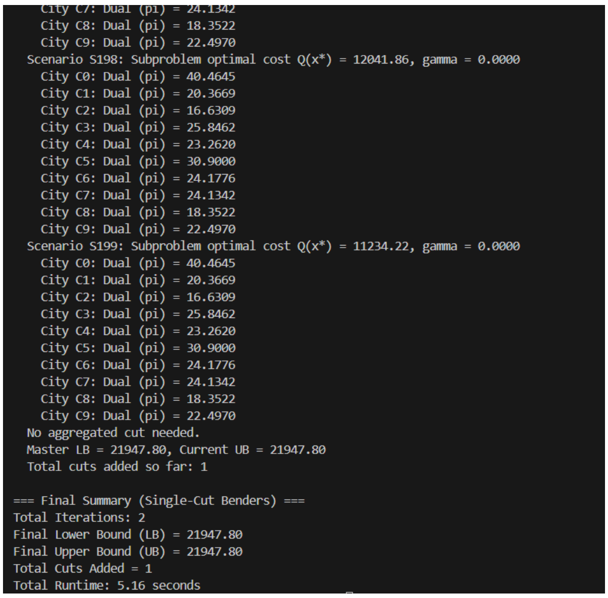
- This script has issues while printing the primal solutions
- This issue is mainly due to the RHS.
- It seems adding a cut with dual, the RHS of the constraint will be consistent
- This script did not implement changes due to time constraints.
- The script is still functional and can be used for testing purposes.
- It seems that presolving the master prob without additional cut from sub prob would give better results.



```
    City C1: Dual for Demand (pi) = 20.3669
    City C2: Dual for Demand (pi) = 16.6309
    City C3: Dual for Demand (pi) = 25.8462
    City C4: Dual for Demand (pi) = 23.2620
    City C5: Dual for Demand (pi) = 30.9000
    City C6: Dual for Demand (pi) = 24.1776
    City C7: Dual for Demand (pi) = 24.1342
    City C8: Dual for Demand (pi) = 18.3522
    City C9: Dual for Demand (pi) = 22.4970
    No cut needed for scenario S198 (n[S198] = 12041.8635 ≥ 12041.86).
  Scenario S199: Subproblem optimal cost Q(x*) = 11234.22
    Dual for Capacity (gamma) = 0.0000
    City C0: Dual for Demand (pi) = 40.4645
    City C1: Dual for Demand (pi) = 20.3669
    City C2: Dual for Demand (pi) = 16.6309
    City C3: Dual for Demand (pi) = 25.8462
    City C4: Dual for Demand (pi) = 23.2620
    City C5: Dual for Demand (pi) = 30.9000
    City C6: Dual for Demand (pi) = 24.1776
    City C7: Dual for Demand (pi) = 24.1342
    City C8: Dual for Demand (pi) = 18.3522
    City C9: Dual for Demand (pi) = 22.4970
    No cut needed for scenario S199 (n[S199] = 11234.2159 ≥ 11234.22).
  Master LB = 21947.80, Current UB = 21947.80
  Cuts added this iteration: 0
  Total cuts added so far: 200

=== Final Summary (Multi-Cut Benders) ===
Total Iterations: 2
Final Lower Bound (LB) = 21947.80
Final Upper Bound (UB) = 21947.80
Total Cuts Added = 200
Total Runtime: 6.15 seconds
PS D:\University of Edinburgh\All coding stuff>
```

Figure 1: Multiple-cut Benders



```
    City C7: Dual (pi) = 24.1342
    City C8: Dual (pi) = 18.3522
    City C9: Dual (pi) = 22.4970
  Scenario S198: Subproblem optimal cost Q(x*) = 12041.86, gamma = 0.0000
    City C0: Dual (pi) = 40.4645
    City C1: Dual (pi) = 20.3669
    City C2: Dual (pi) = 16.6309
    City C3: Dual (pi) = 25.8462
    City C4: Dual (pi) = 23.2620
    City C5: Dual (pi) = 30.9000
    City C6: Dual (pi) = 24.1776
    City C7: Dual (pi) = 24.1342
    City C8: Dual (pi) = 18.3522
    City C9: Dual (pi) = 22.4970
  Scenario S199: Subproblem optimal cost Q(x*) = 11234.22, gamma = 0.0000
    City C0: Dual (pi) = 40.4645
    City C1: Dual (pi) = 20.3669
    City C2: Dual (pi) = 16.6309
    City C3: Dual (pi) = 25.8462
    City C4: Dual (pi) = 23.2620
    City C5: Dual (pi) = 30.9000
    City C6: Dual (pi) = 24.1776
    City C7: Dual (pi) = 24.1342
    City C8: Dual (pi) = 18.3522
    City C9: Dual (pi) = 22.4970
  No aggregated cut needed.
  Master LB = 21947.80, Current UB = 21947.80
  Total cuts added so far: 1

=== Final Summary (Single-Cut Benders) ===
Total Iterations: 2
Final Lower Bound (LB) = 21947.80
Final Upper Bound (UB) = 21947.80
Total Cuts Added = 1
Total Runtime: 5.16 seconds
```

Figure 2: Single-cut Benders

## 2 Q2: Stochastic Program with SAA

This question addresses a two-stage stochastic program where uncertainty is modeled by a Poisson distribution. Due to the nature of the expectation involving this distribution, exact evaluation is often intractable, motivating the use of Sample Average Approximation (SAA).

### 2.1 Problem Formulation

The optimization problem is defined as:

$$\min_{0\leq x\leq 5}\{f(x) := -0.75x + \mathbb{E}_\xi[Q(x,\xi)]\}$$

where the random variable $\xi$ follows a Poisson distribution with mean $\lambda = 0.5$, denoted $\xi \sim \text{Poisson}(0.5)$. The function $Q(x,\xi)$ represents the optimal value of the second-stage (recourse) problem for a given first-stage decision $x$ and a realization $\xi$ of the random variable:

$$Q(x,\xi) = \min_{v_1,v_2,v_3,v_4\geq 0}\{-v_1 + 3v_2 + v_3 + v_4\}$$

subject to the linear constraints:

$$\begin{aligned} -v_1 + v_2 - v_3 + v_4 &= \xi + 0.5x \\ -v_1 + v_2 + v_3 - v_4 &= 1 + \xi + 0.25x \end{aligned}$$

The first-stage decision involves selecting $x \in [0,5]$, incurring a direct cost of $-0.75x$. The second-stage decision involves selecting non-negative $v_1, v_2, v_3, v_4$ to minimize the recourse cost, adapting to the realized value of $\xi$ and the chosen $x$. The overall objective is to minimize the sum of the first-stage cost and the *expected* second-stage cost.

### 2.2 Sample Average Approximation (SAA) Rationale

The core challenge lies in evaluating the term $\mathbb{E}_\xi[Q(x,\xi)]$, which represents the expected value of the recourse function over the Poisson(0.5) distribution. Since the Poisson distribution has infinite support (all non-negative integers), calculating this expectation directly would require summing an infinite series, where each term involves solving the second-stage linear program $Q(x,k)$ for $k = 0,1,2,\ldots$. This is computationally impractical.

SAA circumvents this difficulty by approximating the true expectation with an average over a finite sample drawn from the distribution. If we draw $N$ independent and identically distributed (i.i.d.) samples $\{\xi^1,\ldots,\xi^N\}$ from Poisson(0.5), the SAA problem is formulated as:

$$\nu_N^* = \min_{0\leq x\leq 5}\left\{\overline{f}_N(x) := -0.75x + \frac{1}{N}\sum_{j=1}^{N} Q(x,\xi^j)\right\}$$

Let $x_N^*$ be the optimal solution to this SAA problem. By the Law of Large Numbers, as $N \to \infty$, the sample average objective function $\overline{f}_N(x)$ converges to the true objective function $f(x)$. Consequently, under suitable technical conditions, the SAA optimal value $\nu_N^*$ converges to the true optimal value $\nu^*$, and the SAA optimal solution $x_N^*$ converges to a true optimal solution $x^*$.

### 2.3 Statistical Lower Bound Estimation

The assignment requires computing a 95

- **Theoretical Foundation:** It is known that the optimal value of the SAA problem is a statistically downward-biased estimator of the true optimal value, i.e., $\mathbb{E}[\nu_N^*] \leq \nu^*$. Therefore, by estimating $\mathbb{E}[\nu_N^*]$, we obtain a statistical lower bound for $\nu^*$.

- **Procedure:**

  1. **Replication:** Perform $M = 10$ independent replications (batches) of the SAA procedure.
  2. **Sampling and Solving per Batch:** For each batch $m = 1, \ldots, M$:
     - Generate an i.i.d. sample $\{\xi^{1,m}, \ldots, \xi^{N,m}\}$ of size $N = 30$ from Poisson(0.5).
     - Formulate and solve the SAA problem corresponding to this sample:

     $$\nu_N^m = \min_{0 \leq x \leq 5} \left\{ -0.75x + \frac{1}{N} \sum_{j=1}^{N} Q(x, \xi^{j,m}) \right\}$$

     This yields the optimal objective value $\nu_N^m$ for batch $m$.
  3. **Calculate Sample Mean:** Compute the average of the optimal values obtained across the $M$ batches:

  $$L_{N,M} = \frac{1}{M} \sum_{m=1}^{M} \nu_N^m$$

  This $L_{N,M}$ serves as our point estimate for $\mathbb{E}[\nu_N^*]$.
  4. **Calculate Sample Standard Deviation:** Compute the sample standard deviation of the $M$ optimal values:

  $$s_L(M) = \sqrt{\frac{1}{M-1} \sum_{m=1}^{M} (\nu_N^m - L_{N,M})^2}$$

  This estimates the variability between the SAA optimal values from different batches.
  5. **Confidence Interval Construction:** Since $M = 10$ is a relatively small number of batches, we use the Student's t-distribution to construct the confidence interval. For a 95

  $$\left[ L_{N,M} - t_{9,0.025} \frac{s_L(M)}{\sqrt{M}}, \quad L_{N,M} + t_{9,0.025} \frac{s_L(M)}{\sqrt{M}} \right]$$

  Let this interval be denoted $[\underline{L}_{N,M}, \overline{L}_{N,M}]$. Since $\mathbb{E}[\nu_N^*] \leq \nu^*$, the interval $[\underline{L}_{N,M}, \overline{L}_{N,M}]$ provides a 95

## 2.4 Statistical Upper Bound Estimation and Optimality Gap

The next step involves selecting a candidate solution $\hat{x}$ and evaluating its performance using a large, independent sample to obtain a 95

- **Theoretical Foundation:** For any feasible first-stage solution $\hat{x} \in [0, 5]$, its true expected cost $f(\hat{x}) = -0.75\hat{x} + \mathbb{E}[Q(\hat{x}, \xi)]$ is, by definition, greater than or equal to the minimum possible true expected cost, $\nu^*$. Thus, $f(\hat{x})$ provides an upper bound on $\nu^*$. We estimate $f(\hat{x})$ using Monte Carlo simulation with a large sample size $\tilde{N}$.

- **Procedure:**

  1. **Candidate Selection:** From the $M = 10$ SAA runs performed for the lower bound estimation, identify the run $m^*$ that yielded the best (minimum) objective value $\nu_N^{m^*} = \min_m \{\nu_N^m\}$. Let the corresponding optimal first-stage solution be $x_N^{m^*}$. This solution is chosen as the candidate solution: $\hat{x} = x_N^{m^*}$.

2. **Generate Large Validation Sample:** Draw a *new*, *independent* sample $\{\tilde{\xi}^1, \ldots, \tilde{\xi}^{\tilde{N}}\}$ of size $\tilde{N} = 500$ from Poisson(0.5). This sample must be independent of those used for the lower bound calculation.
3. **Evaluate Candidate Solution:** Estimate the true performance of $\hat{x}$ by calculating its average objective value over the large validation sample:

$$U_{\tilde{N}}(\hat{x}) = \frac{1}{\tilde{N}} \sum_{j=1}^{\tilde{N}} f(\hat{x}, \tilde{\xi}^j) = -0.75\hat{x} + \frac{1}{\tilde{N}} \sum_{j=1}^{\tilde{N}} Q(\hat{x}, \tilde{\xi}^j)$$

   This requires solving the second-stage LP $Q(\hat{x}, \tilde{\xi}^j)$ for each of the $\tilde{N} = 500$ scenarios in the validation sample. $U_{\tilde{N}}(\hat{x})$ is an unbiased estimator of $f(\hat{x})$.
4. **Calculate Sample Standard Deviation:** Compute the sample standard deviation of the evaluated objective values $f(\hat{x}, \tilde{\xi}^j)$:

$$s_U(\tilde{N}) = \sqrt{\frac{1}{\tilde{N}-1} \sum_{j=1}^{\tilde{N}} (f(\hat{x}, \tilde{\xi}^j) - U_{\tilde{N}}(\hat{x}))^2}$$

   This estimates the variability of the objective function value for the fixed solution $\hat{x}$ across different realizations of $\xi$.
5. **Confidence Interval Construction:** Since $\tilde{N} = 500$ is large, we can use the standard normal distribution (z-distribution) via the Central Limit Theorem. For a 95

$$\left[ U_{\tilde{N}}(\hat{x}) - z_{0.025} \frac{s_U(\tilde{N})}{\sqrt{\tilde{N}}}, \quad U_{\tilde{N}}(\hat{x}) + z_{0.025} \frac{s_U(\tilde{N})}{\sqrt{\tilde{N}}} \right]$$

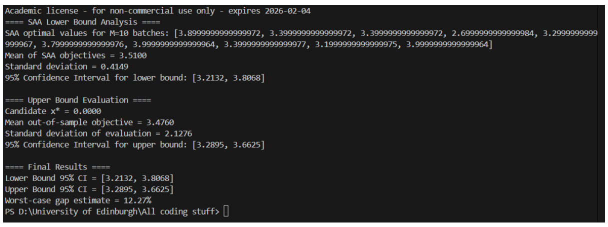
   Let this interval be $[\underline{U}_{\tilde{N}}, \overline{U}_{\tilde{N}}]$. Since $f(\hat{x}) \geq \nu^*$, this interval provides a 95

- **Optimality Gap Estimation:** The SAA procedure provides statistical bounds, not deterministic ones. The estimated (worst-case) optimality gap is the difference between the upper confidence limit for the upper bound and the lower confidence limit for the lower bound:

$$\text{Estimated Gap} = \overline{U}_{\tilde{N}} - \underline{L}_{N,M}$$

  This value represents the width of the interval within which the true optimal value $\nu^*$ is estimated to lie, with approximately 95

## 2.5 Results



```
Academic license - for non-commercial use only - expires 2026-02-04
==== SAA Lower Bound Analysis ====
SAA optimal values for M=10 batches: [3.8999999999999972, 3.3999999999999972, 3.3999999999999972, 2.6999999999999984, 3.2999999999
999967, 3.7999999999999976, 3.9999999999999964, 3.3999999999999977, 3.1999999999999975, 3.9999999999999964]
Mean of SAA objectives = 3.5100
Standard deviation = 0.4149
95% Confidence Interval for lower bound: [3.2132, 3.8068]

==== Upper Bound Evaluation ====
Candidate x* = 0.0000
Mean out-of-sample objective = 3.4760
Standard deviation of evaluation = 2.1276
95% Confidence Interval for upper bound: [3.2895, 3.6625]

==== Final Results ====
Lower Bound 95% CI = [3.2132, 3.8068]
Upper Bound 95% CI = [3.2895, 3.6625]
Worst-case gap estimate = 12.27%
PS D:\University of Edinburgh\All coding stuff>
```

Figure 3: SAA

The code was ran with various random seeds and the confidence interval kept on changing. Now you can refer to the figure using 3.

# 3 Q3: Problem-Driven Scenario Reduction

This question requires applying the Cost-Space Scenario Clustering (CSSC) method from Keutchayan et al. (2023) [1] to the stochastic problem defined in Q2. The aim is to reduce a large set of $N = 100$ scenarios to a smaller representative set of $N' = 10$ scenarios and compare the quality of the solution obtained from the reduced set against the solution from the larger set.

## 3.1 Rationale for Problem-Driven Reduction

Traditional scenario reduction methods often cluster scenarios based on proximity in the parameter space (e.g., using Euclidean distance between scenario vectors $\xi_i$). Examples include k-means, k-medoids, or moment matching. While intuitive, these methods ignore the structure of the optimization problem itself. Two scenarios that are far apart in parameter space might induce very similar optimal second-stage costs or decisions, while two scenarios close in parameter space might lead to vastly different outcomes [1].

Problem-driven methods, like CSSC, aim to overcome this by incorporating information from the optimization problem, typically related to costs or solutions, into the clustering process [1]. The CSSC method specifically focuses on clustering scenarios based on the similarity of their cost functions $F(x, \xi_i)$, evaluated across a relevant set of first-stage solutions [1]. The idea is that if the cost functions associated with two scenarios behave similarly for critical first-stage decisions, these scenarios can be grouped together, even if the scenario parameters themselves are dissimilar [1]. This aims to minimize the *implementation error* – the loss incurred by using the solution from the reduced problem in the context of the original, full set of scenarios [1].

Unlike distribution-based clustering (e.g., k-means applied directly to the $\xi$ values), CSSC incorporates the problem's cost structure via the opportunity-cost matrix $V$. By minimizing cost discrepancies within clusters based on this matrix, CSSC aims to find a reduced set that better preserves the objective function landscape, leading to more informed scenario reductions aligned with the optimization goal, rather than just statistical proximity [1].

## 3.2 CSSC Algorithm Steps for Q3

Following the assignment instructions and the CSSC methodology [1]:

1. **Generate Initial Sample ($\mathcal{S}$):** Draw $N = 100$ i.i.d. scenarios $\{\xi^1, \ldots, \xi^{100}\}$ from the Poisson(0.5) distribution, as used in Q2. These are assumed to be equiprobable ($p_i = 1/N$).

2. **Compute Opportunity-Cost Matrix ($V$):** This is Step 1 of the CSSC algorithm [1].

   - For each scenario $i = 1, \ldots, N$: Solve the *single-scenario* deterministic problem to find an optimal first-stage solution $x_i^*$:

   $$x_i^* \in \arg\min_{0 \leq x \leq 5} \{F(x, \xi^i) := -0.75x + Q(x, \xi^i)\}$$

   - For each pair $(i, j)$ where $i, j \in \{1, \ldots, N\}$: Evaluate the total cost of using solution $x_i^*$ under scenario $\xi^j$. This defines the matrix element $V_{i,j} = F(x_i^*, \xi^j)$. Computing $Q(x_i^*, \xi^j)$ requires solving the second-stage LP for the fixed $x_i^*$ and scenario $\xi^j$. This step involves solving $N = 100$ single-scenario optimization problems and $N^2 = 10,000$ second-stage LPs (though $N$ of these are already solved when finding $x_i^*$). Parallel computation can significantly speed this up [1].

3. **Solve Clustering MIP:** This is Step 2 of the CSSC algorithm [1]. Set up and solve the following MIP (Eqs. (24)-(29) in [1]) to partition the $N = 100$ scenarios into $K = N' = 10$

clusters, identifying a representative scenario $j$ for each cluster:

$$
\begin{aligned}
\min \quad & \frac{1}{N}\sum_{j=1}^{N} t_j \\
\text{s.t.} \quad & t_j \geq \sum_{i=1}^{N} x_{ij}V_{j,i} - \sum_{i=1}^{N} x_{ij}V_{j,j} && \forall j \in \{1,\dots,N\} \\
& t_j \geq \sum_{i=1}^{N} x_{ij}V_{j,j} - \sum_{i=1}^{N} x_{ij}V_{j,i} && \forall j \in \{1,\dots,N\} \\
& \sum_{j=1}^{N} x_{ij} = 1 && \forall i \in \{1,\dots,N\} \\
& x_{ij} \leq u_j && \forall i,j \in \{1,\dots,N\} \\
& x_{jj} = u_j && \forall j \in \{1,\dots,N\} \\
& \sum_{j=1}^{N} u_j = K && (K = 10) \\
& x_{ij} \in \{0,1\} && \forall i,j \in \{1,\dots,N\} \\
& u_j \in \{0,1\} && \forall j \in \{1,\dots,N\} \\
& t_j \geq 0 && \forall j \in \{1,\dots,N\}
\end{aligned}
$$

- $u_j = 1$ if scenario $j$ is selected as a representative.
- $x_{ij} = 1$ if scenario $i$ is assigned to the cluster represented by scenario $j$.
- $t_j$ measures the discrepancy within the cluster represented by $j$.
- The objective minimizes the average discrepancy across the chosen representatives.
- Constraints ensure each scenario belongs to exactly one cluster, the representative belongs to its own cluster, and exactly $K = 10$ representatives are chosen.

4. **Form Reduced Sample ($\mathcal{S}'$):** The solution to the MIP gives $K = 10$ indices $j^*$ for which $u_{j^*} = 1$. The reduced set $\mathcal{S}'$ consists of the corresponding scenarios $\{\xi^{j_1^*}, \dots, \xi^{j_{10}^*}\}$. The probability $p'_k$ for each representative scenario $\xi^{j_k^*}$ in $\mathcal{S}'$ is set to $|C_k|/N$, where $|C_k|$ is the number of original scenarios $i$ assigned to representative $j_k^*$ (i.e., number of $i$ for which $x_{i,j_k^*} = 1$) [1].

5. **Solve SP with $\mathcal{S}$ and $\mathcal{S}'$:**
   - Solve the SAA problem using the full initial sample $\mathcal{S}$ (N=100 scenarios, probabilities $1/N$): Find $x \in \arg\min_{0\leq x\leq 5}\{-0.75x + (1/N)\sum_{i=1}^{N} Q(x,\xi^i)\}$.
   - Solve the SAA problem using the reduced sample $\mathcal{S}'$ (N'=10 scenarios, probabilities $p'_k$): Find $x' \in \arg\min_{0\leq x\leq 5}\{-0.75x + \sum_{k=1}^{N'} p'_k Q(x,\xi^{j_k^*})\}$.

6. **Compare Solutions:** Evaluate the "true" objective value for both solutions $x$ and $x'$ using a large, independent validation sample of $N_{val} = 10,000$ scenarios $\{\xi_l^{val}\}_{l=1}^{N_{val}}$ drawn from Poisson(0.5):
   - True value of $x$: $f_{true}(x) = -0.75x + \frac{1}{N_{val}}\sum_{l=1}^{N_{val}} Q(x,\xi_l^{val})$.
   - True value of $x'$: $f_{true}(x') = -0.75x' + \frac{1}{N_{val}}\sum_{l=1}^{N_{val}} Q(x',\xi_l^{val})$.

   Compare $f_{true}(x)$ and $f_{true}(x')$ to assess how well the solution $x'$ from the reduced set approximates the solution $x$ from the larger set in terms of expected performance. Ideally, $f_{true}(x')$ should be very close to $f_{true}(x)$.

### 3.3 Result

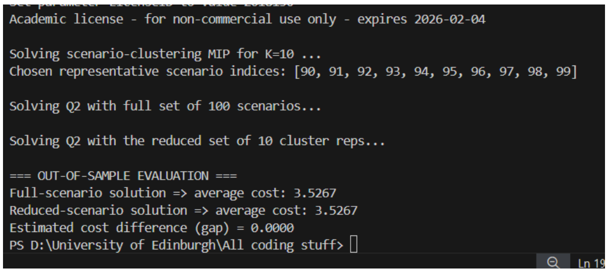



Figure 4: Q3

# 4 Conclusions

- **Q1 (Benders Decomposition):** The principles of Benders decomposition (both multi-cut and single-cut variants) were detailed for solving the two-stage stochastic inventory problem. The method leverages the problem's structure by decomposing it into a master problem for first-stage decisions and scenario-specific subproblems for recourse actions, iteratively adding optimality or feasibility cuts derived from subproblem duals until convergence. The comparison highlights the trade-off between the number of cuts added per iteration and the size/complexity of the master problem.

- **Q2 (Sample Average Approximation):** The SAA method was described to handle a two-stage problem with Poisson uncertainty. Procedures for obtaining statistical lower bounds (via multiple SAA replications) and upper bounds (by evaluating a candidate solution on a large validation set), along with their respective 95% confidence intervals, were detailed. The optimality gap, derived from these confidence intervals, provides a measure of solution quality.

- **Q3 (Problem-Driven Scenario Reduction):** The Cost-Space Scenario Clustering (CSSC) method from Keutchayan et al. (2023) [1] was presented as a problem-driven alternative to purely distribution-based scenario reduction. The steps involved – computing an opportunity-cost matrix based on single-scenario solutions and solving a clustering MIP to select representatives – were outlined in the context of applying it to the Q2 problem. The rationale is to create a reduced scenario set that better preserves the cost structure relevant to the optimization problem, aiming to minimize implementation error.

Collectively, these questions illustrate powerful techniques for tackling the computational challenges posed by uncertainty in optimization problems, ranging from exploiting problem structure via decomposition to approximating expectations via sampling and intelligently reducing the number of scenarios considered.

## 4.1 Gen AI usage:

- For report writing, generative AI tools such as ChatGPT were used to obtain a template for LaTeX code from the provided Word document which included a draft of the writeup. Furthermore, Grammarly was utilized for spell checking and paraphrasing in the word document.

- ChatGPT was utilized to neatly organise, add spacing where required, and also include descriptive comments according to the Word document in the codes.

Note: Benders.py and workshop 3,4 solutions were referred to while writing the code, building the logic and finishing this assignment. I would like to thank Dr. M. Bodur for uploading Gurobi examples on Learn; this assignment would not have been possible without them.

# References

[1] J. Keutchayan, J. Ortmann, and W. Rei. Problem-driven scenario clustering in stochastic optimization. s *Computational Management Science*, 20(1):13, 2023. (Referenced via: `s10287-023-00446-2.pdf`)

[2] M. Bodur. MATH 11247: Benders Decomposition (Chapter 3 Slides). University of Edinburgh, 2025. (Referenced via: `Slides_Chapter3_Benders.pdf`)

[3] M. Bodur. MATH 11247: Sampling (Chapter 4 Slides). University of Edinburgh, 2025. (Referenced via: `Slides_Chapter4_Sampling.pdf`)

## A Appendix: Key Variable Glossary

### Q1: Benders Variables

- $x_n$: (1st stage) Inventory allocated to city $n$.
- $\eta_k$: (MP, Multi-Cut) Approx. cost for scenario $k$.
- $\theta$: (MP, Single-Cut) Approx. total expected recourse cost.
- $u_n^k, v_n^k$: (SP $k$) Corrective shipments for city $n$.
- $z_n^k, s_n^k$: (SP $k$) Leftover inventory / shortage for city $n$.
- $\pi_{cap}^k, \pi_{dem,n}^k$: (SP $k$, Dual) Multipliers for capacity and demand constraints.

### Q2: SAA Variables

- $x$: (1st stage) Decision variable ($0 \leq x \leq 5$).
- $\xi$: Random variable, $\xi \sim$ Poisson(0.5).
- $v_1, \ldots, v_4$: (2nd stage) Recourse variables for a given $\xi$.
- $v_1^j, \ldots, v_4^j$: (SAA formulation) 2nd stage variables for scenario $j$.
- $\nu^*$: True optimal objective value.
- $\nu_N^*$: Optimal value of SAA problem with $N$ samples.
- $L_{N,M}$: Sample mean of $\nu_N^*$ over $M$ batches (LB estimate).
- $U_{\tilde{N}}(\hat{x})$: Sample mean performance of candidate $\hat{x}$ over $\tilde{N}$ samples (UB estimate).

### Q3: CSSC Variables

- $\xi^1, \ldots, \xi^N$: Initial set of $N = 100$ scenarios.
- $x_i^*$: Optimal $x$ for single-scenario problem $i$.
- $V_{i,j}$: Opportunity cost: $F(x_i^*, \xi^j)$.
- $u_j$: (MIP) Binary; 1 if scenario $j$ is a representative.
- $x_{ij}$: (MIP) Binary; 1 if scenario $i$ assigned to representative $j$.
- $t_j$: (MIP) Cluster discrepancy measure for representative $j$.
- $\mathcal{S}'$: Reduced set of $N' = 10$ representative scenarios.
- $p_k'$: Probability of representative scenario $k$ in $\mathcal{S}'$.
- $x, x'$: Optimal solutions from SAA using $\mathcal{S}$ and $\mathcal{S}'$, respectively.

# B Code

## B.1 Q1

### B.1.1 Multicut

```python
"""
Assignment 3 Q1 - Multi-Cut Benders Decomposition
Note: 1) This script has issues while printing the primal solutions
      2) This issue is mainly due to the RHS.
      3) It seems adding a cut with dual, the RHS of the constraint will be consistent
      4) The changes were not implemented in this script due to time constraints.
      5) The script is still functional and can be used for testing purposes.
      6) It seems that presolving the master prob without additional cut from sub prob would give better results.
This script implements the multi-cut version of Benders decomposition for Assignment 3 Q1.
It splits the original problem into a master (first-stage) problem and many subproblems (one per scenario).
If the current master solution underestimates the recourse function in any scenario, a separate Benders cut
is generated and added for that scenario.
"""
import gurobipy as gp
from gurobipy import GRB, LinExpr, quicksum
import time
import ReadData as Data

# --------------------------
# Data & Model Parameters
# --------------------------
# Load all required data:
# - 'cities' is the list of cities (e.g., ['C0', 'C1', ...]).
# - 'scenarios' is the list of scenario keys (e.g., ['S0', 'S1', ...]).
# - 'theta' holds the first-stage cost for each city.
# - 'theta_s' contains second-stage cost coefficients (for decisions u and v).
# - 'h' and 'g' are the penalties for unused inventory (z) and shortage (s) respectively.
# - 'I' is the total available inventory at the center.
# - 'Yn' holds the initial inventory at each city.
# - 'demand' provides the demand for each (city, scenario) pair.
# - 'prob' is the (uniform) probability assigned to each scenario.
cities    = Data.cities
scenarios = Data.scenarios
theta     = Data.theta
theta_s   = Data.theta_s
h         = Data.h
g         = Data.g
I         = Data.I
Yn        = Data.Yn
demand    = Data.demand
prob      = Data.prob

# Set tolerance for determining cut violation and the maximum allowed iterations.
CutViolationTolerance = 1e-4
max_iters = 200

# --------------------------
# Build the Master Problem
# --------------------------
```

```python
# The master problem decides on the first-stage variables x for each city.
MP = gp.Model("Master_Multicut")
MP.Params.OutputFlag = 0  # Turn off Gurobi logging for clarity.
MP.modelSense = GRB.MINIMIZE

# Create decision variables x[c] for each city, with cost theta[c].
x = {c: MP.addVar(lb=0, obj=theta[c], name=f"x_{c}") for c in cities}

# Add a constraint for the total inventory available at the center.
MP.addConstr(quicksum(x[c] for c in cities) <= I, name="CenterInventory")

# For every scenario, create an epigraph variable n_k that will approximate the recourse cost Q_k(x).
n_vars = {k: MP.addVar(lb=0, obj=prob, name=f"n_{k}") for k in scenarios}

MP.update()

# -------------------------
# Build the Subproblem for Each Scenario
# -------------------------
def build_subproblem(model_name):
    """
    Build an empty subproblem model for a given scenario.

    For each city, four decision variables are created:
      - u: adjustment variable linked to capacity.
      - v: adjustment variable linked to demand.
      - z: unused inventory.
      - s: shortage.

    The objective minimizes the cost of adjustments (weighted by theta_s)
    plus the penalties for unused inventory (h) and shortage (g).
    """
    sp = gp.Model(model_name)
    sp.Params.OutputFlag = 0
    u_vars = {c: sp.addVar(lb=0, name=f"u_{c}") for c in cities}
    v_vars = {c: sp.addVar(lb=0, name=f"v_{c}") for c in cities}
    z_vars = {c: sp.addVar(lb=0, name=f"z_{c}") for c in cities}
    s_vars = {c: sp.addVar(lb=0, name=f"s_{c}") for c in cities}
    sp.setObjective(
        quicksum(theta_s[c]*(u_vars[c] + v_vars[c]) for c in cities) +
        h * quicksum(z_vars[c] for c in cities) +
        g * quicksum(s_vars[c] for c in cities),
        GRB.MINIMIZE
    )
    sp.update()
    return sp, u_vars, v_vars, z_vars, s_vars

# Create and store a subproblem for each scenario.
SP = {}         # Dictionary for subproblem models.
sub_vars = {}   # Dictionary for subproblem decision variable dictionaries.
for k in scenarios:
    sp, u_vars, v_vars, z_vars, s_vars = build_subproblem(f"SP_{k}")
    SP[k] = sp
    sub_vars[k] = (u_vars, v_vars, z_vars, s_vars)

# -------------------------
# Function to Solve a Subproblem
# -------------------------
def solve_subproblem(k, xsol):
    """
    Solve the subproblem for scenario k given the current first-stage solution (xsol).
```

```

    First, old constraints are removed and then the subproblem is updated:
      (A) A capacity constraint: I + sum(u) must be at least the sum of v and the current xsol.
      (B) For each city c, a demand balance constraint is enforced:
          Yn[c] + xsol[c] + v[c] + s[c] equals demand[(c,k)] + z[c] + u[c].

    After solving, the function returns:
      - SPobj: the optimal objective value (recourse cost) for the scenario.
      - gamma: dual multiplier for the capacity constraint.
      - pi: a dictionary of dual multipliers for each city's demand constraint.
    """
    sp = SP[k]
    u_vars, v_vars, z_vars, s_vars = sub_vars[k]
    # Remove previous constraints to update with the new xsol.
    if len(sp.getConstrs()) > 0: # Check if constraints exist before removing
        sp.remove(sp.getConstrs())
        sp.update() # Update after removal
    # (A) Capacity constraint.
    cap_constr = sp.addConstr(
        I + quicksum(u_vars[c] for c in cities) >= quicksum(v_vars[c] for c in cities) + sum(xsol[c] for c in cities),
        name="Capacity"
    )
    # (B) Demand constraints for each city.
    demand_constr = {}
    for c in cities:
        demand_constr[c] = sp.addConstr(
            Yn[c] + xsol[c] + v_vars[c] + s_vars[c] == demand[(c, k)] + z_vars[c] + u_vars[c],
            name=f"Demand_{c}"
        )
    sp.update() # Update after adding constraints
    sp.optimize()
    if sp.status != GRB.OPTIMAL:
        print(f"Subproblem for scenario {k} infeasible or failed (Status: {sp.status})!")
        return None, None, None
    SPobj = sp.objVal
    # Get dual values from the constraints.
    gamma = cap_constr.Pi
    pi = {c: demand_constr[c].Pi for c in cities}
    return SPobj, gamma, pi

# --------------------------
# Main Benders Decomposition Loop (Multi-Cut)
# --------------------------
TotalCutsAdded = 0   # Total number of cuts added.
iteration = 0        # Iteration counter.
BestUB = float('inf')  # Best (lowest) upper bound encountered.
LB = -float('inf')     # Initialize lower bound
start_time = time.time()

while iteration < max_iters:
    iteration += 1
    MP.update() # Make sure model reflects added cuts before solving
    MP.optimize()
    if MP.status != GRB.OPTIMAL:
        print("Master problem infeasible or failed!")
        break
    LB = MP.objVal  # The current lower bound from the master problem.

    # Get current solution for first-stage decisions and epigraph variables.
```

```
    xsol = {c: x[c].X for c in cities}
    nsol = {k: n_vars[k].X for k in scenarios}

    # Calculate an upper bound: first-stage cost + weighted recourse costs.
    current_UB = sum(theta[c]*xsol[c] for c in cities)
    subproblem_data = {} # Store results for cut generation
    all_subproblems_ok = True

    print(f"\n--- Iteration {iteration} ---")
    print(f"  Current LB = {LB:.4f}")
    # print("  Current first-stage solution:") # Optional: Verbose x printing
    # for c in cities:
    #     print(f"     x[{c}] = {xsol[c]:.4f}")

    print("  Solving subproblems...")
    for k in scenarios:
        SPobj, gamma, pi = solve_subproblem(k, xsol)
        if SPobj is None:
            all_subproblems_ok = False
            break # Need to handle failure (e.g., add feasibility cut if appropriate)
        current_UB += prob * SPobj
        subproblem_data[k] = (SPobj, gamma, pi) # Store for cut generation
        # print(f"    Scenario {k}: Q(x*) = {SPobj:.2f}, gamma = {gamma:.4f}") # Optional

    if not all_subproblems_ok:
        print("  Subproblem solve failed. Stopping.")
        break

    # Update best UB
    if current_UB < BestUB:
        BestUB = current_UB
    print(f"  Current UB = {current_UB:.4f}")
    print(f"  Best UB Found So Far = {BestUB:.4f}")

    # Check for convergence
    gap = BestUB - LB
    if gap <= CutViolationTolerance * max(1, abs(LB)):
        print(f"\nConvergence achieved: UB-LB gap ({gap:.4f}) within tolerance.")
        break

    # Generate and add cuts
    cuts_added_this_iter = 0
    print("  Checking for violated cuts...")
    for k in scenarios:
        SPobj, gamma, pi = subproblem_data[k]

        # If the current epigraph variable underestimates the recourse cost, add a Benders cut.
        if nsol[k] < SPobj - CutViolationTolerance:
            # Build the cut: n_vars[k] - sum{(pi[c] + gamma)*x[c]} >= SPobj - sum{(pi[c] + gamma)*xsol[c]}
            lhs = LinExpr(n_vars[k])
            rhs = SPobj
            for c in cities:
                coeff = pi[c] + gamma
                lhs.addTerms(-coeff, x[c])
                rhs -= coeff * xsol[c]
            MP.addConstr(lhs >= rhs, name=f"BendersCut_{k}_iter{iteration}")
            # print(f"    Added Benders cut for scenario {k}") # Optional verbose output
```

```
                TotalCutsAdded += 1
                cuts_added_this_iter += 1
            # else:
                # print(f"    No cut needed for scenario {k}") # Optional verbose
    output

        print(f"  Cuts added this iteration: {cuts_added_this_iter}")

        # If no new cuts were added, the solution should be optimal (within
    tolerance).
        if cuts_added_this_iter == 0:
            print("\nNo cuts added in this iteration. Converged.")
            break

# --- End of loop ---
end_time = time.time()
total_time = end_time - start_time

# ---------------------------
# Final Summary
# ---------------------------
print("\n=== Final Summary (Multi-Cut Benders) ===")
print(f"Termination Reason: {'Converged' if cuts_added_this_iter == 0 else ('
    Max Iterations Reached' if iteration >= max_iters else 'Error')}")
print(f"Total Iterations: {iteration}")
print(f"Final Lower Bound (LB) = {LB:.4f}")
print(f"Best Upper Bound (UB) = {BestUB:.4f}")
# Final Gap Calculation
final_gap_abs = BestUB - LB
final_gap_rel = (final_gap_abs / max(1e-10, abs(BestUB)) * 100) if BestUB !=
    float('inf') else float('inf')
print(f"Final Optimality Gap = {final_gap_abs:.4f} ({final_gap_rel:.4f}%)")
print(f"Total Cuts Added = {TotalCutsAdded}")
print(f"Total Runtime: {total_time:.2f} seconds")

# Print final optimal x solution if available
if 'xsol' in locals():
    print("\nOptimal first-stage solution (x):")
    for c in cities:
        print(f"  x[{c}] = {xsol[c]:.4f}")
else:
    print("\nNo final solution obtained.")
```

Listing 1: Python Code for Q1 Implementation (Multi-Cut Benders)

#### B.1.2 Singlecut

```
"""
Assignment 3 Q1 - Single-Cut Benders Decomposition
Note: 1) This script has issues while printing the primal solutions
      2) This issue is mainly due to the RHS.
      3) It seems adding a cut with dual, the RHS of the constraint will be
    consistent
      4) The changes were not implemented in this script due to time
    constraints.
      5) The script is still functional and can be used for testing purposes.
      6) It seems that presolving the master prob without additional cut from
    sub prob would give better results.
This script implements the single-cut version of Benders decomposition for
    Assignment 3 Q1.
Unlike the multi-cut approach, here we aggregate the dual information from all
    scenarios and
```

```python
add one consolidated Benders cut per iteration.
"""

import gurobipy as gp
from gurobipy import GRB, LinExpr, quicksum
import time
import ReadData as Data # Assuming ReadData.py is accessible

# -------------------------
# Data & Model Parameters
# -------------------------
# Load necessary data from the ReadData module
cities    = Data.cities
scenarios = Data.scenarios
theta     = Data.theta      # 1st stage cost coeff for x[c]
theta_s   = Data.theta_s    # 2nd stage cost coeff for u[c], v[c]
h         = Data.h          # 2nd stage cost for z[c]
g         = Data.g          # 2nd stage cost for s[c]
I         = Data.I          # Central capacity
Yn        = Data.Yn         # Initial city inventory
demand    = Data.demand     # Demand d[c,k]
prob      = Data.prob       # Scenario probability

# Tolerance and iteration limit
CutViolationTolerance = 1e-4
max_iters = 200

# -------------------------
# Build the Master Problem (Single-Cut Version)
# -------------------------
MP = gp.Model("Master_Singlecut")
MP.Params.OutputFlag = 0
MP.modelSense = GRB.MINIMIZE

# First-stage variables x[c]
x = {c: MP.addVar(lb=0, obj=theta[c], name=f"x_{c}") for c in cities}
# Single epigraph variable theta_var approximating sum p_k * Q_k(x)
theta_var = MP.addVar(lb=0, obj=1.0, name="theta") # Coefficient is 1

# Central inventory constraint
MP.addConstr(quicksum(x[c] for c in cities) <= I, name="CenterInventory")
MP.update()

# -------------------------
# Build the Subproblem Structure (Template)
# -------------------------
# This function is identical to the one in the multi-cut version
def build_subproblem(model_name):
    """ Creates an empty Gurobi model for a scenario subproblem. """
    sp = gp.Model(model_name)
    sp.Params.OutputFlag = 0
    u_vars = {c: sp.addVar(lb=0, name=f"u_{c}") for c in cities}
    v_vars = {c: sp.addVar(lb=0, name=f"v_{c}") for c in cities}
    z_vars = {c: sp.addVar(lb=0, name=f"z_{c}") for c in cities}
    s_vars = {c: sp.addVar(lb=0, name=f"s_{c}") for c in cities}
    sp.setObjective(
        quicksum(theta_s[c]*(u_vars[c] + v_vars[c]) for c in cities) +
        h * quicksum(z_vars[c] for c in cities) +
        g * quicksum(s_vars[c] for c in cities),
        GRB.MINIMIZE
    )
    sp.update()
    return sp, u_vars, v_vars, z_vars, s_vars
```

```

# Create subproblem instances
SP = {}
sub_vars = {}
for k in scenarios:
    sp, u, v, z, s = build_subproblem(f"SP_{k}")
    SP[k] = sp
    sub_vars[k] = (u, v, z, s)

# --------------------------
# Function to Solve a Single Subproblem
# --------------------------
# This function is identical to the one in the multi-cut version
def solve_subproblem(k, xsol):
    """ Solves subproblem k for given xsol; returns obj, gamma, pi. """
    sp = SP[k]
    u_vars, v_vars, z_vars, s_vars = sub_vars[k]
    if len(sp.getConstrs()) > 0:
        sp.
\subsection{Q2}
\begin{lstlisting}[language=Python, caption={Python Code for Q2 Implementation
    (SAA)}, label={lst:q2code}]
import math
import random
import numpy as np
from gurobipy import Model, GRB, quicksum
from scipy.stats import t # Make sure scipy is imported if using t.ppf

##############################################################################
# DATA AND PROBLEM DESCRIPTION
##############################################################################
# In this two-stage stochastic program, our uncertain parameter xi follows a
# Poisson distribution with mean 0.5. The first-stage decision variable x is in
# [0, 5]. Once xi is realized, we solve the second-stage LP:
#
#   Q(x, xi) = min { -v1 + 3*v2 + v3 + v4 }
#              s.t.  -v1 + v2 - v3 + v4 = xi + 0.5*x,
#                    -v1 + v2 + v3 - v4 = 1 + xi + 0.25*x,
#                    v1, v2, v3, v4 >= 0.
#
# The full objective is: min -0.75*x + E[ Q(x, xi) ]
#
# SAA is used: M=10 batches of N=30 samples for LB CI. Best x evaluated
# on N_tilde=500 samples for UB CI.

##############################################################################
# FUNCTION: Solve the SAA Extensive Form for One Batch
##############################################################################
def solve_SAA_problem(xi_sample):
    """
    Build and solve the extensive form for a given xi sample (size N).
    Returns: optimal objective value, optimal x.
    """
    N = len(xi_sample)
    model = Model("SAA")
    model.setParam("OutputFlag", 0) # Suppress Gurobi output

    # First-stage variable x
    x = model.addVar(lb=0, ub=5, name="x", vtype=GRB.CONTINUOUS)

    # Second-stage variables v1..v4 for each scenario i
    v1 = model.addVars(N, lb=0, name="v1")
    v2 = model.addVars(N, lb=0, name="v2")
```

```python
    v3 = model.addVars(N, lb=0, name="v3")
    v4 = model.addVars(N, lb=0, name="v4")

    # Objective: first-stage + average recourse
    obj_expr = -0.75 * x
    for i in range(N):
        obj_expr += (1.0 / N) * (-v1[i] + 3 * v2[i] + v3[i] + v4[i])
    model.setObjective(obj_expr, GRB.MINIMIZE)

    # Constraints per scenario
    for i in range(N):
        xi_val = xi_sample[i]
        model.addConstr(-v1[i] + v2[i] - v3[i] + v4[i] == xi_val + 0.5 * x)
        model.addConstr(-v1[i] + v2[i] + v3[i] - v4[i] == 1 + xi_val + 0.25 * x)

    model.optimize()
    # Handle potential infeasibility or other statuses if necessary
    if model.status == GRB.OPTIMAL:
        return model.objVal, x.X
    else:
        print(f"Warning: SAA problem did not solve to optimality (Status: {model.status})")
        return float('inf'), None # Or handle error appropriately

##############################################################################
# FUNCTION: Evaluate a Candidate Solution Out-of-Sample
##############################################################################
def evaluate_candidate(x_candidate, xi_eval):
    """
    For fixed x_candidate, evaluate performance over xi_eval samples.
    Solves second-stage LP for each sample to get Q(x_candidate, xi).
    Returns mean and stdev of total cost F = -0.75*x + Q.
    """
    recourse_values = []
    N_eval = len(xi_eval)

    for i in range(N_eval):
        xi_val = xi_eval[i]
        ssp = Model("subproblem")
        ssp.setParam("OutputFlag", 0)

        v1 = ssp.addVar(lb=0, name="v1")
        v2 = ssp.addVar(lb=0, name="v2")
        v3 = ssp.addVar(lb=0, name="v3")
        v4 = ssp.addVar(lb=0, name="v4")

        ssp.setObjective(-v1 + 3 * v2 + v3 + v4, GRB.MINIMIZE)

        ssp.addConstr(-v1 + v2 - v3 + v4 == xi_val + 0.5 * x_candidate)
        ssp.addConstr(-v1 + v2 + v3 - v4 == 1 + xi_val + 0.25 * x_candidate)

        ssp.optimize()
        # Handle potential infeasibility if the problem doesn't have complete recourse
        if ssp.status == GRB.OPTIMAL:
            recourse_values.append(ssp.objVal)
        else:
            print(f"Warning: Subproblem evaluation failed for xi={xi_val} (Status: {ssp.status})")
            recourse_values.append(float('inf')) # Or handle error

    # Calculate total costs and stats
```

```python
    total_costs = [-0.75 * x_candidate + rv for rv in recourse_values if rv !=
    float('inf')]
    if not total_costs: # Handle case where all subproblems failed
        return float('inf'), float('nan')

    mean_total_cost = np.mean(total_costs)
    stdev_total_cost = np.std(total_costs, ddof=1) if len(total_costs) > 1 else
     0

    return mean_total_cost, stdev_total_cost

##############################################################################
# MAIN PROCEDURE: SAA and Out-of-Sample Evaluation
##############################################################################
def main():
    import statistics # Use statistics module for mean/stdev

    # --- SAA Settings ---
    M = 10          # Batches
    N = 30          # Samples per batch
    N_tilde = 500   # Out-of-sample size
    seed_base = 160325
    poisson_lambda = 0.5
    alpha = 0.05    # For 95% CI

    # Step 1: Solve M SAA problems for Lower Bound estimation
    results_saa_obj = []
    results_saa_x = []
    best_obj_batch = float('inf')
    best_x_candidate = None

    print(f"Running {M} SAA batches (N={N})...")
    for m in range(M):
        seed = seed_base + m
        np.random.seed(seed)
        xi_sample = np.random.poisson(poisson_lambda, N)

        obj_val, x_val = solve_SAA_problem(xi_sample)
        if x_val is not None: # Check if solve was successful
            results_saa_obj.append(obj_val)
            results_saa_x.append(x_val)
            if obj_val < best_obj_batch:
                best_obj_batch = obj_val
                best_x_candidate = x_val
        # Optional: Add handling if solve_SAA_problem fails multiple times

    if not results_saa_obj:
        print("Error: No SAA batches solved successfully.")
        return

    # Calculate LB CI
    LB_mean = statistics.mean(results_saa_obj)
    # Use stdev only if M > 1
    LB_stdev = statistics.stdev(results_saa_obj) if M > 1 else 0
    t_crit = t.ppf(1 - alpha / 2, df=M - 1) if M > 1 else float('inf') # Use t-
    dist for small M
    LB_halfwidth = t_crit * LB_stdev / math.sqrt(M) if M > 0 else 0
    LB_CI = (LB_mean - LB_halfwidth, LB_mean + LB_halfwidth)

    print("\n==== SAA Lower Bound Analysis ====")
    # print(f"SAA optimal objectives: {results_saa_obj}") # Can be long
    print(f"Mean of SAA objectives (LB estimate) = {LB_mean:.6f}")
    print(f"Std Dev of SAA objectives = {LB_stdev:.6f}")
```

```python
    print(f"95% CI for Lower Bound: [{LB_CI[0]:.6f}, {LB_CI[1]:.6f}]")

    # Step 2: Evaluate best candidate solution out-of-sample for Upper Bound
    if best_x_candidate is None:
        print("\nError: Could not determine a candidate solution.")
        return

    print(f"\nEvaluating candidate x* = {best_x_candidate:.6f} out-of-sample (
    N_tilde={N_tilde})...")
    np.random.seed(99999) # Use a different seed for evaluation
    xi_eval = np.random.poisson(poisson_lambda, N_tilde)
    mean_eval, stdev_eval = evaluate_candidate(best_x_candidate, xi_eval)

    # Calculate UB CI (using normal approx z=1.96 for large N_tilde)
    z_crit = 1.96
    UB_halfwidth = z_crit * stdev_eval / math.sqrt(N_tilde) if N_tilde > 0 else
     0
    UB_CI = (mean_eval - UB_halfwidth, mean_eval + UB_halfwidth)

    print("\n==== Upper Bound Evaluation ====")
    print(f"Mean out-of-sample objective (UB estimate) = {mean_eval:.6f}")
    print(f"Std Dev of out-of-sample objectives = {stdev_eval:.6f}")
    print(f"95% CI for Upper Bound: [{UB_CI[0]:.6f}, {UB_CI[1]:.6f}]")

    # Step 3: Compute worst-case optimality gap
    # Gap = Upper end of UB CI - Lower end of LB CI
    worst_case_gap = UB_CI[1] - LB_CI[0]

    print("\n==== Final Results ====")
    print(f"Lower Bound 95% CI = [{LB_CI[0]:.6f}, {LB_CI[1]:.6f}]")
    print(f"Upper Bound 95% CI = [{UB_CI[0]:.6f}, {UB_CI[1]:.6f}]")
    print(f"Worst-case optimality gap estimate = {worst_case_gap:.6f}")
    # Optional: Relative gap, e.g., gap / abs(LB_mean) if LB_mean is non-zero

if __name__ == "__main__":
    main()
```

Listing 2: Python Code for Q1 Implementation (Single-Cut Benders)

### B.2 Q2

```python
import math
import random
import numpy as np
from gurobipy import Model, GRB, quicksum

###############################################################################
# DATA AND PROBLEM DESCRIPTION
###############################################################################
# In this two-stage stochastic program, our uncertain parameter xi follows a
# Poisson distribution with mean 0.5. The first-stage decision variable x is in
# [0, 5] and represents an investment (or similar decision). Once xi is
   realized,
# we solve the second-stage linear program:
#
#   Q(x, xi) = min { -v1 + 3*v2 + v3 + v4 }
#             s.t.   -v1 + v2 - v3 + v4 = xi + 0.5*x,
#                    -v1 + v2 + v3 - v4 = 1 + xi + 0.25*x,
#                    v1, v2, v3, v4 >= 0.
#
# The full objective is to minimize:
#       -0.75*x + E[ Q(x, xi) ]
```

```
#
# We approximate the expectation using Sample Average Approximation (SAA). Specifically:
#   - For each batch (with N = 30 scenarios), we solve an extensive form.
#   - We repeat this for M = 10 batches to estimate a lower bound and its 95% CI.
#   - We then select the best candidate x (the one with the lowest SAA objective)
#     and evaluate it on a large independent sample (N_tilde = 500) to form an upper bound.

##############################################################################
# FUNCTION: Solve the SAA Extensive Form for One Batch
##############################################################################
def solve_SAA_problem(xi_sample):
    """
    Build and solve the extensive form of the two-stage problem for a given set of
    xi samples (of size N). The model is formulated as:

      minimize   -0.75*x + (1/N)*sum_{i=1}^N (-v1_i + 3*v2_i + v3_i + v4_i)
      subject to, for each scenario i:
                 -v1_i + v2_i - v3_i + v4_i = xi_sample[i] + 0.5*x,
                 -v1_i + v2_i + v3_i - v4_i = 1 + xi_sample[i] + 0.25*x,
                 v1_i, v2_i, v3_i, v4_i >= 0,
                 and 0 <= x <= 5.

    Returns:
      - The optimal objective value for this batch.
      - The optimal value of the first-stage decision x.
    """
    N = len(xi_sample)
    model = Model("SAA")
    model.setParam("OutputFlag", 0)  # Suppress solver output for cleaner logs

    # Define first-stage decision variable x (continuous between 0 and 5)
    x = model.addVar(lb=0, ub=5, name="x", vtype=GRB.CONTINUOUS)

    # Define second-stage decision variables for each scenario i
    v1 = model.addVars(N, lb=0, name="v1")
    v2 = model.addVars(N, lb=0, name="v2")
    v3 = model.addVars(N, lb=0, name="v3")
    v4 = model.addVars(N, lb=0, name="v4")

    # Build the objective function: first-stage cost plus average recourse cost
    obj_expr = -0.75 * x
    for i in range(N):
        obj_expr += (1.0 / N) * (-v1[i] + 3 * v2[i] + v3[i] + v4[i])
    model.setObjective(obj_expr, GRB.MINIMIZE)

    # Add constraints for each scenario i using the corresponding xi value
    for i in range(N):
        xi_val = xi_sample[i]
        model.addConstr(-v1[i] + v2[i] - v3[i] + v4[i] == xi_val + 0.5 * x)
        model.addConstr(-v1[i] + v2[i] + v3[i] - v4[i] == 1 + xi_val + 0.25 * x)

    # Solve the model and return the results
    model.optimize()
    return model.objVal, x.X

##############################################################################
# FUNCTION: Evaluate a Candidate Solution Out-of-Sample
```

```python
################################################################################
def evaluate_candidate(x_candidate, xi_eval):
    """
    For a fixed candidate x value, evaluate its performance over a large set of out-of-
    sample scenarios (xi_eval). For each scenario, we solve the second-stage problem:

        Q(x_candidate, xi) = min { -v1 + 3*v2 + v3 + v4 }
        subject to:
            -v1 + v2 - v3 + v4 = xi + 0.5*x_candidate,
            -v1 + v2 + v3 - v4 = 1 + xi + 0.25*x_candidate,
            v1, v2, v3, v4 >= 0.

    The function returns the mean and standard deviation of the total cost (first-stage plus
    recourse cost) across the out-of-sample scenarios.
    """
    recourse_values = []  # To store the recourse cost for each scenario
    from gurobipy import Model, GRB

    # Loop over every scenario in the evaluation set
    for xi_val in xi_eval:
        ssp = Model("subproblem")
        ssp.setParam("OutputFlag", 0)

        # Define recourse variables for this scenario
        v1 = ssp.addVar(lb=0, name="v1")
        v2 = ssp.addVar(lb=0, name="v2")
        v3 = ssp.addVar(lb=0, name="v3")
        v4 = ssp.addVar(lb=0, name="v4")

        # Set the second-stage objective for this scenario
        ssp.setObjective(-v1 + 3 * v2 + v3 + v4, GRB.MINIMIZE)

        # Add constraints for the current scenario
        ssp.addConstr(-v1 + v2 - v3 + v4 == xi_val + 0.5 * x_candidate)
        ssp.addConstr(-v1 + v2 + v3 - v4 == 1 + xi_val + 0.25 * x_candidate)

        ssp.optimize()
        recourse_values.append(ssp.objVal)

    # Compute the average second-stage cost and its variability
    avg_Q = np.mean(recourse_values)
    stdev_Q = np.std(recourse_values, ddof=1)
    total_costs = [-0.75 * x_candidate + rv for rv in recourse_values]
    return np.mean(total_costs), np.std(total_costs, ddof=1)

################################################################################
# MAIN PROCEDURE: SAA and Out-of-Sample Evaluation
################################################################################
def main():
    import statistics
    import math
    from scipy.stats import t

    # --- SAA Settings ---
    M = 10          # Number of independent SAA batches
    N = 30          # Number of scenarios per batch (sample size)
    N_tilde = 500   # Number of out-of-sample scenarios for candidate evaluation
    seed_base = 160325   # Base seed for reproducibility

    # Step 1: Solve M independent SAA problems and record the optimal objective
```

```
values
    results_saa = []
    best_obj = float('inf')
    best_x = None

    for m in range(M):
        # Use a unique seed for each batch to ensure independent samples
        seed = seed_base + m
        np.random.seed(seed)
        xi_sample = np.random.poisson(0.5, N)

        # Solve the extensive form for the current batch
        obj_val, x_val = solve_SAA_problem(xi_sample)
        results_saa.append(obj_val)

        # Update candidate if current batch gives a lower objective
        if obj_val < best_obj:
            best_obj = obj_val
            best_x = x_val

    # Compute the lower bound estimate and its 95% confidence interval
    LB_mean = statistics.mean(results_saa)
    LB_stdev = statistics.stdev(results_saa) if M > 1 else statistics.pstdev(results_saa)
    t_val = t.ppf(0.975, df=M-1)
    LB_halfwidth = t_val * LB_stdev / math.sqrt(M)
    LB_CI = (LB_mean - LB_halfwidth, LB_mean + LB_halfwidth)

    print("==== SAA Lower Bound Analysis ====")
    print(f"SAA optimal values for M={M} batches: {results_saa}")
    print(f"Mean of SAA objectives = {LB_mean:.4f}")
    print(f"Standard deviation = {LB_stdev:.4f}")
    print(f"95% Confidence Interval for lower bound: [{LB_CI[0]:.4f}, {LB_CI[1]:.4f}]")

    # Step 2: Out-of-sample evaluation of the best candidate x*
    cand_x = best_x
    np.random.seed(99999)  # Fixed seed for reproducible evaluation
    xi_eval = np.random.poisson(0.5, N_tilde)
    mean_eval, stdev_eval = evaluate_candidate(cand_x, xi_eval)

    # Use normal approximation (due to large N_tilde) to form 95% CI for the candidate evaluation
    UB_halfwidth = 1.96 * (stdev_eval / math.sqrt(N_tilde))
    UB_CI = (mean_eval - UB_halfwidth, mean_eval + UB_halfwidth)

    print("\n==== Upper Bound Evaluation ====")
    print(f"Candidate x* = {cand_x:.4f}")
    print(f"Mean out-of-sample objective = {mean_eval:.4f}")
    print(f"Standard deviation of evaluation = {stdev_eval:.4f}")
    print(f"95% Confidence Interval for upper bound: [{UB_CI[0]:.4f}, {UB_CI[1]:.4f}]")

    # Step 3: Compute the worst-case optimality gap (as a percentage)
    worst_gap_num = UB_CI[1] - LB_CI[0]   # Difference between upper bound (high end) and lower bound (low end)
    worst_gap_den = max(abs(UB_CI[1]), 1e-6)
    worst_gap_pct = 100.0 * worst_gap_num / worst_gap_den

    print("\n==== Final Results ====")
    print(f"Lower Bound 95% CI = [{LB_CI[0]:.4f}, {LB_CI[1]:.4f}]")
    print(f"Upper Bound 95% CI = [{UB_CI[0]:.4f}, {UB_CI[1]:.4f}]")
    print(f"Worst-case gap estimate = {worst_gap_pct:.2f}%")
```

```

if __name__ == "__main__":
    main()
```

Listing 3: Python Code for Q2 Imp

### B.3 Q3

```
import numpy as np
import gurobipy as gp
from gurobipy import GRB

################################################################################
# Q3: PROBLEM-DRIVEN SCENARIO REDUCTION
#
# This script illustrates how to reduce an original scenario set (size N=100,
  from
# Poisson(0.5)) to a smaller set (size N'=10). We refer back to Q2's two-stage
# model:
#
#   min_{0 <= x <= 5} -0.75*x + E_xi[ Q(x, xi) ],
#
# where xi ~ Poisson(0.5), and
#
#   Q(x, xi) = min_{v1,v2,v3,v4 >= 0} [ -v1 + 3v2 + v3 + v4 ]
#              s.t. -v1 + v2 - v3 + v4 = xi + 0.5*x
#                   -v1 + v2 + v3 - v4 = 1 + xi + 0.25*x.
#
# The approach:
#   1) Generate N=100 scenarios {xi_full}.
#   2) For each scenario i, solve the single-scenario Q2 subproblem => x_i^*.
#   3) Construct cost matrix V[i,j] = total cost if we "prepare" for xi_i
#      but actually face xi_j.
#   4) Solve a scenario-clustering MIP (from the reference paper's eqs. (24)
  -(29))
#      that picks N'=10 cluster representatives.
#   5) Solve the original two-stage problem with all 100 scenarios =>
  x_full_sol,
#      and with only the 10 cluster reps => x_sub_sol.
#   6) Compare out-of-sample performance with Mtest=10,000 fresh draws from
  Poisson(0.5).
################################################################################

def solve_single_scenario_subproblem(xi_val):
    """
    For Q2: Solve the single-scenario version of the 2-stage model for demand
    xi_val.
    That is:
      min   -0.75*x + [ -v1 + 3*v2 + v3 + v4 ]
      s.t.  0 <= x <= 5
            -v1 + v2 - v3 + v4 = xi_val + 0.5*x
            -v1 + v2 + v3 - v4 = 1 + xi_val + 0.25*x
            v1,v2,v3,v4 >= 0
    Returns x_i^*, the best first-stage decision if xi_val were the only
    scenario.
    """
    m = gp.Model("single_scenario")
    m.setParam("OutputFlag", 0)

    # First-stage decision variable
    x = m.addVar(lb=0, ub=5, name="x", vtype=GRB.CONTINUOUS)

```

```python
    # Second-stage recourse variables for Q2
    v1 = m.addVar(lb=0, name="v1")
    v2 = m.addVar(lb=0, name="v2")
    v3 = m.addVar(lb=0, name="v3")
    v4 = m.addVar(lb=0, name="v4")

    # Add scenario-based constraints:
    m.addConstr(-v1 + v2 - v3 + v4 == xi_val + 0.5*x,  name="constr1")
    m.addConstr(-v1 + v2 + v3 - v4 == 1 + xi_val + 0.25*x, name="constr2")

    # Full objective: first-stage cost + second-stage cost
    #  = -0.75*x + (-v1 + 3*v2 + v3 + v4)
    obj_expr = -0.75*x + (-v1 + 3*v2 + v3 + v4)
    m.setObjective(obj_expr, GRB.MINIMIZE)

    m.optimize()
    # Return the optimal x value
    return x.X

def evaluate_cost(x_val, xi_val):
    """
    For Q2: Evaluate the total cost if x=x_val is chosen and scenario xi_val
    occurs.
    cost(x_val, xi_val) = -0.75*x_val + Q(x_val, xi_val).
    We compute Q(...) by solving a small LP (the second-stage problem).
    """
    m = gp.Model("evaluate")
    m.setParam("OutputFlag", 0)

    # Recourse variables for scenario xi_val
    v1 = m.addVar(lb=0, name="v1")
    v2 = m.addVar(lb=0, name="v2")
    v3 = m.addVar(lb=0, name="v3")
    v4 = m.addVar(lb=0, name="v4")

    # Constraints
    m.addConstr(-v1 + v2 - v3 + v4 == xi_val + 0.5*x_val,  name="constr1")
    m.addConstr(-v1 + v2 + v3 - v4 == 1 + xi_val + 0.25*x_val, name="constr2")

    # Second-stage objective: -v1 + 3*v2 + v3 + v4
    m.setObjective(-v1 + 3*v2 + v3 + v4, GRB.MINIMIZE)
    m.optimize()

    recourse_cost = m.objVal
    total_cost = -0.75*x_val + recourse_cost
    return total_cost

def build_cost_matrix(xi_array):
    """
    Build the NxN cost matrix V, where N = len(xi_array).
      Step 1) For i in [0..N-1], solve single-scenario subproblem => x_i^*.
      Step 2) For each i, evaluate cost if scenario j occurs => V[i,j].
    Thus, V[i,j] = cost( x_i^*, xi_array[j] ).
    """
    N = len(xi_array)
    x_star = np.zeros(N)  # store x_i^*

    # 1) Solve single-scenario subproblem for each scenario i
    for i in range(N):
        x_star[i] = solve_single_scenario_subproblem(xi_array[i])

    # 2) Evaluate cost with each scenario j
    V = np.zeros((N,N))
```

```
    for i in range(N):
        for j in range(N):
            V[i,j] = evaluate_cost(x_star[i], xi_array[j])
    return V

def solve_clustering_MIP(V, K):
    """
    Solve the scenario-clustering MIP from the Q3 reference (Eqs. (24)-(29)).
    - N = total scenarios
    - K = number of clusters
    The MIP picks K scenario "representatives" and partitions the N scenarios among them,
    minimizing a cost-based discrepancy measure.

    Returns:
      rep_scenarios: list of scenario indices chosen as cluster reps
      assignment[i]: scenario i is assigned to cluster rep assignment[i]
    """
    N = V.shape[0]
    model = gp.Model("ScenarioClustering")
    model.setParam("OutputFlag", 0)

    # u_j = 1 if scenario j is a cluster representative
    u = model.addVars(N, vtype=GRB.BINARY, name="u")

    # x_{i,j} = 1 if scenario i is assigned to rep j
    xij = model.addVars(N, N, vtype=GRB.BINARY, name="xij")

    # t_j >= 0 for each cluster representative j
    t = model.addVars(N, lb=0, name="t")

    # Objective: sum of t_j / N
    obj = gp.quicksum(t[j] for j in range(N)) / N
    model.setObjective(obj, GRB.MINIMIZE)

    # For each j, linearize the absolute difference:
    #   t_j >= sum_i xij[i,j]*(V[j,i] - V[j,j]) and t_j >= -(...).
    for j in range(N):
        lhs = gp.quicksum(xij[i,j]*V[j,i] for i in range(N)) - gp.quicksum(xij[i,j]*V[j,j] for i in range(N))
        model.addConstr(t[j] >= lhs)
        model.addConstr(t[j] >= -lhs)

    # Each scenario i must be assigned to exactly one cluster j
    for i in range(N):
        model.addConstr(gp.quicksum(xij[i,j] for j in range(N)) == 1)

    # If j is not chosen as rep => xij[i,j] = 0, also xij[j,j] = u_j
    for j in range(N):
        for i in range(N):
            model.addConstr(xij[i,j] <= u[j])
        model.addConstr(xij[j,j] == u[j])

    # Exactly K representatives
    model.addConstr(gp.quicksum(u[j] for j in range(N)) == K)

    model.optimize()

    # Extract solution
    rep_scenarios = []
    assignment = np.zeros(N, dtype=int)
    for j in range(N):
        if u[j].X > 0.5:
```

```
            rep_scenarios.append(j)

    for i in range(N):
        for j in range(N):
            if xij[i,j].X > 0.5:
                assignment[i] = j
                break

    return rep_scenarios, assignment

def solve_stochastic_program(xi_array, prob_array):
    """
    Solve the Q2 two-stage problem in an extensive-form style, but for a
    smaller set of M scenarios:
      min    -0.75*x + sum_m [ prob_array[m] * recourse_cost(...) ]
      s.t.   0 <= x <= 5
            second-stage constraints for each scenario m.

    We'll replicate the constraints for each scenario and unify the first-stage
     x.
    Return x_stoch, the optimal first-stage solution.
    """
    M = len(xi_array)
    bigm = gp.Model("StochEF")
    bigm.setParam("OutputFlag", 0)

    # First-stage variable x (0 <= x <= 5)
    x = bigm.addVar(lb=0, ub=5, name="x")

    # Second-stage vars for each scenario
    v1 = {}
    v2 = {}
    v3 = {}
    v4 = {}
    for m in range(M):
        v1[m] = bigm.addVar(lb=0, name=f"v1_{m}")
        v2[m] = bigm.addVar(lb=0, name=f"v2_{m}")
        v3[m] = bigm.addVar(lb=0, name=f"v3_{m}")
        v4[m] = bigm.addVar(lb=0, name=f"v4_{m}")

    # Add constraints for each scenario
    for m in range(M):
        xi_val = xi_array[m]
        # -v1[m] + v2[m] - v3[m] + v4[m] = xi_val + 0.5*x
        bigm.addConstr(-v1[m] + v2[m] - v3[m] + v4[m] == xi_val + 0.5*x)
        # -v1[m] + v2[m] + v3[m] - v4[m] = 1 + xi_val + 0.25*x
        bigm.addConstr(-v1[m] + v2[m] + v3[m] - v4[m] == 1 + xi_val + 0.25*x)

    # Build objective
    # sum_{m} prob_array[m]*(-v1[m] + 3v2[m] + v3[m] + v4[m]) + (-0.75*x)
    obj_expr = -0.75*x
    for m in range(M):
        obj_expr += prob_array[m]*(-v1[m] + 3*v2[m] + v3[m] + v4[m])
    bigm.setObjective(obj_expr, GRB.MINIMIZE)

    bigm.optimize()
    return x.X

def main():
    # Step A: Generate N=100 Poisson(0.5) scenarios
    np.random.seed(160325)
    N = 100
    xi_full = np.random.poisson(lam=0.5, size=N)
```

```python
    p_full = np.ones(N)/N

    # Step B: Build cost matrix V[i,j] where i= "trained scenario", j="actual scenario"
    print("Building NxN cost matrix from single-scenario solutions...")
    V = build_cost_matrix(xi_full)

    # Step C: Solve scenario clustering MIP => pick K=10 representatives
    K = 10
    print(f"\nSolving scenario-clustering MIP for K={K} ...")
    rep_scenarios, assignment = solve_clustering_MIP(V, K)
    print("Chosen representative scenario indices:", rep_scenarios)

    # Step D: Solve the full Q2 problem with all 100 scenarios
    print("\nSolving Q2 with full set of 100 scenarios...")
    x_full_sol = solve_stochastic_program(xi_full, p_full)

    # Step E: Build the smaller scenario set S' (xi_sub) from the cluster reps
    xi_sub = []
    prob_sub = []
    for j in rep_scenarios:
        # Count how many i are assigned to j
        cluster_size = sum(1 for i in range(N) if assignment[i] == j)
        # Probability p_j = cluster_size / N
        p_j = float(cluster_size)/N
        xi_sub.append(xi_full[j])
        prob_sub.append(p_j)
    xi_sub = np.array(xi_sub)
    prob_sub = np.array(prob_sub)

    # Solve Q2 with the reduced set of 10 scenarios
    print(f"\nSolving Q2 with the reduced set of {K} cluster reps...")
    x_sub_sol = solve_stochastic_program(xi_sub, prob_sub)

    # Step F: Out-of-sample evaluation: Mtest=10,000 fresh Poisson samples
    Mtest = 10_000
    np.random.seed(99999)
    xi_test = np.random.poisson(0.5, size=Mtest)

    # Evaluate x_full_sol out-of-sample
    total_full = 0.0
    for s in range(Mtest):
        total_full += evaluate_cost(x_full_sol, xi_test[s])
    total_full /= Mtest

    # Evaluate x_sub_sol out-of-sample
    total_sub = 0.0
    for s in range(Mtest):
        total_sub += evaluate_cost(x_sub_sol, xi_test[s])
    total_sub /= Mtest

    # Final comparison
    print("\n=== OUT-OF-SAMPLE EVALUATION ===")
    print(f"Full-scenario solution => average cost: {total_full:.4f}")
    print(f"Reduced-scenario solution => average cost: {total_sub:.4f}")
    gap_est = total_sub - total_full
    print(f"Estimated cost difference (gap) = {gap_est:.4f}")

if __name__ == "__main__":
    main()
```

Listing 4: Python Code for Q3 Implementation (CSSC)