UNIVERSITY OF PISA
Department of Computer Science
Master in Data Science and Business Informatics

Master Thesis

Problem-based Scenario Reduction in Two-stage Stochastic Optimization

Supervisors:
Prof. Antonio Frangioni
Dr. Duy Nghi Benoît Tran

Student:
Minh Duc Pham

Academic Year 2025/2026

Abstract

This thesis focuses on applying the Cost-Space Scenario Clustering (CSSC) method to Two-stage Stochastic Optimization, with computational experiments conducted on the Capacitated Facility Location Problem (CFLP) and the Unit Commitment Problem (UCP), specifically within Energy Community (EC) formulations. The CSSC method clusters scenarios based on their structural impact on the objective function and optimal decisions. This is achieved by constructing an opportunity-cost matrix and subsequently solving a Mixed-Integer Linear Programming (MILP) partitioning problem to select the optimal representative scenarios. To integrate this methodology into the SMS++ ecosystem, the approach was implemented as a fully problem-agnostic `CSSCScenarioReduction Solver`, supported by a generic `ScenarioReductionBlock` communication layer. This architecture ensures complete independence from any specific problem type and is accompanied by a comprehensive test suite for both applications. Experimental results on standard OR-Lib benchmark instances demonstrate that in most configurations, CSSC yields a lower optimality gap than heuristic methods, with exceptions occurring at large N and aggressive reduction ratios, albeit at a higher computational cost. Ultimately, this work validates the efficacy of the cost-space approach to scenario reduction and contributes a production-ready, generic implementation to the SMS++ library, successfully extending its optimization capabilities to diverse problem domains.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Capacitated Facility Location problem is one of the most important models in operations research, computer science and supply chain management. It has a wide range of applications, such as the site selection or configuration of factories, schools, warehouses, supermarkets, hospitals, stations, and service agents. However, today's global environment is increasingly unpredictable. These factors lead to significant uncertainties in customer demand and transportation costs. Consequently, stochastic optimization has become a vital tool for providing data-driven decision support, helping managers make robust strategic choices in an uncertain world.

Similarly, the Unit Commitment (UC) problem is a fundamental model in power systems engineering and energy economics. It has a wide range of operational applications, such as the short-term scheduling, startup/shutdown decisions, and optimal dispatch of power plants, microgrids, and localized energy communities. However, today's transition toward sustainable energy makes power grid management increasingly unpredictable. The rapid integration of intermittent renewable energy sources, like wind and solar, along with fluctuating customer demands, leads to significant uncertainties in generation capacity and net load. Consequently, Stochastic Unit Commitment (SUC) has become a vital tool for providing reliable operational decisions, helping grid managers maintain power balance and robust asset scheduling in an uncertain environment.

To manage these uncertainties, scenario-based stochastic optimization is commonly used. However, this approach often suffers from the "*curse of dimensionality*." As the variety and number of uncertain variables increase, the computational complexity grows exponentially, making it difficult to solve within a reasonable time. This is where Scenario Reduction (SR) plays a crucial role. The primary goal of SR is to identify a smaller, representative set of scenarios to replace the original large set. This reduces the calculation speed and complexity while ensuring that the optimality gap remains minimized.

This thesis proposes a problem-based approach for the Two-stage Stochastic Optimization. In particular, we will conduct experiments in Capacitated Facility Location Problem (CFLP) and the Two-stage Unit Commiment Problem (UCP) using Cost-Space Scenario Clustering (CSSC). In this method, we do not care if the reduced scenarios look statistically similar to the original ones. Instead, we focus on the "*cost space*". This means

we group scenarios together if they produce similar costs or impacts on our optimization problem. The main goal of CSSC is to minimize the "*implementation error*." This is the error that happens when we apply a solution from a simplified model to the real-world problem. By clustering scenarios based on their costs rather than their statistical values, we can simplify the problem significantly. This approach helps us reduce calculation time while keeping the results accurate for strategic decisions.

The implementation phase of this thesis is carried out using SMS++, a Structured Modeling System based on modern C++ designed for modeling and solving complex mathematical optimization problems. While the system includes various general-purpose tools, specialized components were required to handle the specific requirements of scenario reduction for the Two-stage Capacitated Facility Location Problem and the Two-stage Unit Commiment Problem.

As a core contribution of this work, several new components were developed and integrated into the SMS++ ecosystem. In particular, CSSCScenarioReductionSolver were developed. This solver implements the proposed CSSC algorithm, providing a problem-based approach to reduce the scenario set size. Unlike generic reduction methods, this solver exploits the cost-space relationships within the CFLP and UCP to maintain a high-quality approximation of the original problem.

To validate the proposed approach, we conducted a series of computational experiments to compare this new solver against existing scenario reduction methods in SMS++. These tests focus on two key metrics: the reduction in computational time and the accuracy of the final solutions, measured by the optimality gap.

This thesis is structured as follows. Chapter 2 outlines the theoretical background of Two-Stage Stochastic Programming, followed by an introduction to Problem-based scenario reduction and Cost-space Scenario Clustering algorithms in Chapter 3. Chapter 4 delineates the architecture of SMS++ and its utilization in computing the Capacitated Facility Location and Unit Commiment problems. Chapters 5 and 6 present the software engineering and implementation aspects, specifically the architectural redesign of `ScenarioReductionBlock`, the refactoring of `ScenarioReductionSolver`, and the integration of the new `CSSCScenario ReductionSolver`. Chapter 7 represents the experimental framework. Next, Chapters 8 and 9 evaluate the performance of these methods, reporting computational experiments on the Capacitated Facility Location and Unit Commitment problems, respectively. Finally, Chapter 10 shows the computational experiments on practical instances provided by the Energy team, where CSSC proves to be a useful method for two-stage stochastic optimization.

Chapter 2

Two-stage Stochastic Programming

2.1 Introduction

Stochastic programming (SP) is a mathematical programming framework used to formulate and solve sequential decision-making problems under uncertainty. It relies on the assumption that the probability distribution of the random parameters is known (possibly inferred from data), and that the information about these parameters becomes available stage by stage. The idea is to optimize an objective function that depends on random variables, often by minimizing expected cost, maximizing expected reward, or satisfying constraints with high probability. Stochastic programming is widely applied in areas such as logistics, finance, and engineering. These problems share a common feature: the effect of a current decision depends on uncertain future outcomes. To evaluate this effect, future decisions must be modeled and optimized while taking into account the additional information revealed over time and the limits imposed by earlier decisions. As a result, the expected impact of a decision is expressed as an integral of a function that is itself defined by an optimization problem. This integral is usually difficult to compute analytically, and standard numerical methods are often impractical because of the high dimensionality involved.

In general, the following stochastic formulation is described as below:

$$\min_{x \in X \subseteq \mathbb{R}^n} \left\{ \mathcal{Z}(x) := \sum_{i=1}^{N} p_i F(x, \xi_i) \right\} \tag{2.1}$$

where $\xi_1, \ldots, \xi_N$ are scenarios taking values in $\mathbb{R}^d$, and $F(x, \xi_i)$ represents the cost associated with the decisions $x \in X$ under scenario ξ_i. For the sake of simplicity in the following developments, we often assume that scenarios are equiprobable, i.e., $p_i = 1/N$ for all $i \in \{1, \ldots, N\}$. This formulation implies that the set of first-stage decisions X is deterministic, meaning the constraints defining X do not depend on the realization of the uncertainty.

The optimal solution to this problem is denoted by x^*, and its corresponding optimal

value by $\mathcal{Z}^*$ (or v^*). It is important to note that there are no inherent restrictions on the number of scenarios (N), the dimension of decisions (n), or the number of random parameters (d). However, in practical settings, N is often so large that solving (2.1) directly becomes computationally prohibitive.

We usually use SP as the framework for modeling those problems. In many cases, SP models take the form of a two-stage problem. In this chapter, we focus on how to solve the two-stage stochastic programming problem in general.

Throughout this thesis, we denote the set of scenarios by $\Omega = \{\xi_1, \ldots, \xi_N\}$ where $N = |\Omega|$ is the total number of scenarios.

2.2 Two-stage stochastic programming

In two-stage stochastic programming, decision variables are divided into two distinct stages based on the timing of the realization of uncertain parameters.

1. **First-stage decisions:** These are "here-and-now" decisions, denoted by $x \in X \subseteq \mathbb{R}^n$. They must be made before the uncertainty is revealed. In the context of facility location, these decisions are often strategic and costly to adjust, such as the selection of site locations.
2. **Second-stage decisions:** These are "wait-and-see" (or recourse) decisions, denoted by $y \in Y$. They are made after a specific scenario ξ_i of the uncertainty has been observed. These decisions allow the system to adjust to the realized data, such as determining transportation flows to meet actual demand.

The goal of the two-stage stochastic program is to minimize the total cost, which consists of the deterministic first-stage cost and the expected cost of the second-stage recourse actions. The general formulation is as follows:

$$\min_{x \in X} \ \mathcal{Z}(x) := f(x) + \mathbb{E}_P[F(x, \xi)] \tag{2.2}$$

When the uncertainty is represented by a finite set of N scenarios $\{\xi_1, \xi_2, \ldots, \xi_N\}$ with corresponding probabilities p_i (where $\sum_{i=1}^{N} p_i = 1$), the expectation in (2.2) can be written as a weighted sum:

$$\min_{x \in X} \ \mathcal{Z}(x) = f(x) + \sum_{i=1}^{N} p_i F(x, \xi_i) \tag{2.3}$$

where:

- $f(x)$: represents the first-stage cost function (e.g., investment or construction costs).
- $F(x, \xi_i)$: is the optimal value of the second-stage problem for a given first-stage decision x and a specific realization ξ_i.

For each scenario ξ_i, the second-stage recourse problem is defined as:

$$F(x, \xi_i) = \min_{y \in Y(x,\xi_i)} g(y, \xi_i) \tag{2.4}$$

where $g(y, \xi_i)$ is the operational cost associated with the recourse decision y under scenario ξ_i, and $Y(x, \xi_i)$ is the set of feasible second-stage decisions which depends on the first-stage choice x.

In this thesis, we assume relatively complete recourse. This means that for any feasible first-stage decision $x \in X$, there always exists a feasible second-stage solution y for every possible scenario ξ_i. This assumption ensures that the problem remains well-defined and solvable across all potential uncertainties.

2.3 Key Optimization Targets

To better understand how two-stage stochastic programming operates, we define several key optimization targets. These definitions are essential for the scenario reduction process discussed in later chapters.

First, we denote $x^*(P)$ and $\mathcal{Z}^*(P)$ as the optimal first-stage decisions and the corresponding optimal objective value for the problem under the full scenario distribution P:

$$x^*(P) = \arg\min_{x \in X} \left\{ f(x) + \sum_{i=1}^{N} p_i F(x, \xi_i) \right\} \tag{2.5}$$

$$\mathcal{Z}^*(P) = f(x^*(P)) + \sum_{i=1}^{N} p_i F(x^*(P), \xi_i) \tag{2.6}$$

In practice, we often evaluate how a specific candidate solution $\hat{x}$ performs against the original distribution P. This is defined as:

$$\mathcal{Z}(\hat{x}, P) = f(\hat{x}) + \sum_{i=1}^{N} p_i F(\hat{x}, \xi_i) \tag{2.7}$$

Furthermore, for the scenario reduction process, it is useful to define a deterministic version of the problem that considers only a single scenario ξ_i. This represents the "perfect hindsight" solution:

$$\min_{x \in X} \mathcal{Z}(x, \xi_i) = f(x) + F(x, \xi_i) \tag{2.8}$$

The optimal first-stage decision for this single scenario is denoted by:

$$x^*(\xi_i) = \arg\min_{x \in X} \{ f(x) + F(x, \xi_i) \} \tag{2.9}$$

In practical applications, the scenarios $\xi_1, \ldots, \xi_N$ can be obtained from various sources, such as historical data (e.g., past demand observations) or generated through sampling from a known multivariate probability distribution using the Sample Average Approximation (SAA) method. Ideally, a large number of scenarios N is desired to accurately capture the underlying uncertainty. However, as N increases, the computational complexity of solving problem (2.3) grows significantly. For many large-scale problems, such as the Capacitated Facility Location Problem with a vast number of scenarios, finding an optimal solution $x^*(P)$ can be beyond the reach of even the most advanced commercial solvers. This motivates the need for scenario reduction: replacing the original set of N scenarios with a smaller representative subset of size $K \ll N$, while preserving the quality of the optimal solution. This is the subject of Chapter 3.

Chapter 3

Scenario Reduction approach

3.1 Introduction

In stochastic optimization, we want to make good decisions when the future is uncertain. We describe this uncertainty using scenarios. Each scenario represents one possible future. For example, a scenario may describe tomorrow's demand, prices, or travel times. In practice, uncertainty is complex. To describe it well, we often need many scenarios. These scenarios may come from historical data or computer simulations. As the number of scenarios increases, the optimization model becomes larger. Each scenario adds more variables and constraints. This makes the problem harder to solve. Modern solvers are powerful, but they have limits. When the number of scenarios is very large, the solver may need too much time or memory. In some cases, it may be hard to find a good solution at all. This is not because the model is wrong, but because it is too large.

Scenario reduction is a way to address this problem. The main idea is simple. Instead of using all scenarios, we select a smaller number of representative ones. These selected scenarios should describe the uncertainty almost as well as the full set. If the reduced set is chosen carefully, the optimal solution of the smaller problem will be close to the optimal solution of the original problem.

Traditionally, scenario reduction methods have been *distribution-driven* (DDSR), relying on statistical similarity such as Euclidean or Wasserstein distances. Because these methods focus solely on input data and ignore the model's objective function and constraints, they are often termed *cost-agnostic*. A critical weakness of DDSR is its potential to discard scenarios that, while statistically similar, lead to vastly different optimal decisions or high penalty costs. This is particularly problematic in industrial systems sensitive to small parameter changes.

To overcome these limitations, recent research has shifted toward Problem-Based Scenario Reduction (PBSR). This approach integrates the optimization model directly into the reduction process by measuring distance in the "cost space" or "decision space" rather than the data space. The primary goal of PBSR is to minimize the *implementation error*, which represents the loss in objective performance when a solution from the reduced model is applied to the original problem. By focusing on "decision applicability," PBSR

ensures that the reduction process is cost-aware and tailored to the specific mathematical structure of the problem.

Several recent methods focus on problem-based and decision-based scenario reduction. These approaches group scenarios that lead to similar decisions in the stochastic optimization model. A notable example is the decision-based clustering method proposed by Hewitt et al. (2021). In this method, each scenario is associated with an induced solution, which represents the decisions made if that scenario were known in advance. Scenarios that produce similar induced solutions are grouped into clusters. A representative scenario, called the medoid, is then selected for each cluster. It provides a good approximation of the decisions associated with the other scenarios in the same group.

Other recent work also uses optimization-based distance measures. Bertsimas and Mundru (2022) propose a scenario reduction method for linear stochastic programs with fixed recourse. Their approach minimizes the second-order Wasserstein distance between the reduced and original scenario distributions. While this method has strong theoretical properties, it can only be applied to a restricted class of problems. In particular, it requires that different scenarios lead to different single-scenario optimal solutions. Earlier studies also introduced clustering ideas based on solution sensitivity and performance measures, such as the forward selection in recourse clusters method proposed by Feng and Ryan (2016), as well as performance-based reduction frameworks studied in Sun et al. (2018). Related ideas appear in the predict-and-optimize approach, where a learning model is used to identify a representative scenario and solve a deterministic problem. Although effective in some cases, this approach relies on strong assumptions and may not capture the full value of stochastic solutions.

In this chapter, we introduce the cost-space scenario clustering (CSSC) algorithm and analyze it in detail.

3.2 Cost-space Scenario Clustering algorithm

The CSSC algorithm was first introduced by Keutchayan et al. (2023). It compares scenarios based on their impact on the objective function of the stochastic optimization problem, through the cost functions $F(\cdot, \xi_i)$ and $F(\cdot, \xi_j)$. The objective is to identify representative scenarios that provide a good approximation of the original stochastic problem. From this perspective, the quality of a reduced scenario set depends on how well it preserves objective values and optimal decisions, rather than on how close scenarios are according to a geometric distance in data space.

To illustrate this idea, consider two scenarios ξ_i and ξ_j that are very different in data space. Although the distance between them may be large, suppose they induce identical values of the cost function for all feasible decisions. In this case, the two scenarios are equivalent from the viewpoint of the optimization problem. They can be grouped into a single cluster and represented by either scenario, with a combined probability mass of $2/N$. This reduction does not change the optimal solution or the optimal objective value. A similar argument applies to more general cases. Consider three scenarios ξ_i, ξ_j, and ξ_k such that the cost function value of ξ_k equals the average of the cost function values of ξ_i and ξ_j. These three scenarios can be clustered together and represented by ξ_k, with an

assigned probability of $3/N$. This clustering preserves the solution of the original problem while removing redundant scenarios. Overall, cost-space scenario clustering focuses on the effect of scenarios on the optimization model rather than on their raw data representation. This approach enables effective scenario reduction without sacrificing solution quality, as it preserves the key information that influences optimal decisions.

Suppose the full problem (2.1) cannot be solved directly. We then seek an approximate problem composed of only $K \ll N$ representative scenarios, drawn from the original set $\{\xi_1, \xi_2, \ldots, \xi_N\}$. Denote the reduced scenario set by $\{\tilde{\xi}_1, \tilde{\xi}_2, \ldots, \tilde{\xi}_K\}$ with corresponding probabilities $\{p_1, p_2, \ldots, p_K\}$ satisfying $\sum_{k=1}^{K} p_k = 1$. The approximate problem is:

$$\min_{x \in X \subseteq \mathbb{R}^n} \; \tilde{f}(x) := \sum_{k=1}^{K} p_k \, F(x, \tilde{\xi}_k), \tag{3.1}$$

with optimal solution $\tilde{x}^*$ and optimal value $\tilde{v}^* := \tilde{f}(\tilde{x}^*)$.

Recall from Chapter 2 that the optimal value of the original full problem is

$$v^* := Z^*(P) = f(x^*) + \mathbb{E}_P[F(x^*, \xi)] = \frac{1}{N} \sum_{i=1}^{N} F(x^*, \xi_i),$$

where we assume equiprobable scenarios $p_i = 1/N$ and where $f(x^*)$ denotes the first-stage cost evaluated at the full-problem optimum x^*. To keep notation consistent with the rest of this chapter, we absorb the first-stage cost into F and write

$$Z(x, P) := \frac{1}{N} \sum_{i=1}^{N} F(x, \xi_i),$$

so that $v^* = Z(x^*, P)$.

The implementation error measures the loss incurred when the solution $\tilde{x}^*$ of the reduced problem is evaluated against the full distribution:

$$Z(\tilde{x}^*, P) - v^* = \frac{1}{N} \sum_{i=1}^{N} F(\tilde{x}^*, \xi_i) - \frac{1}{N} \sum_{i=1}^{N} F(x^*, \xi_i) \;\geq\; 0. \tag{3.2}$$

This quantity is always non-negative by optimality of x^*, and equals zero if $\tilde{x}^*$ is also optimal for the full problem.

We now derive an upper bound on the implementation error. To shorten notation, write $f(x) \equiv Z(x, P)$ and $\tilde{f}(x) \equiv Z(x, \tilde{P})$ for the full and reduced objectives respectively. We then bound $|\tilde{f}(\tilde{x}^*) - v^*|$ as follows.

Remark 1. *Throughout the derivation below, $f(x)$ denotes the full-distribution objective $Z(x, P) = \frac{1}{N} \sum_{i=1}^{N} F(x, \xi_i)$, while $\tilde{f}(x)$ denotes the reduced-distribution objective $Z(x, \tilde{P}) = \sum_{k=1}^{K} p_k F(x, \tilde{\xi}_k)$. This notation is local to this section and should not be confused with the first-stage cost function $f(x)$ used in Chapter 2.*

$$\begin{aligned}
\left|\tilde{f}(\tilde{x}^*) - v^*\right| &= \left|\tilde{f}(\tilde{x}^*) - f(x^*)\right| && (3.3)\\
&= \left|\tilde{f}(\tilde{x}^*) - \tilde{f}(\tilde{x}^*) + \tilde{f}(\tilde{x}^*) - f(x^*)\right| \\
&= \left|f(\tilde{x}^*) - \tilde{f}(\tilde{x}^*) + \tilde{f}(\tilde{x}^*) - f(x^*)\right| && (3.4)\\
&\leq \left|f(\tilde{x}^*) - \tilde{f}(\tilde{x}^*)\right| + \left|\tilde{f}(\tilde{x}^*) - f(x^*)\right|. && (3.5)
\end{aligned}$$

We now bound each of the two terms on the right-hand side of (3.5) separately.

Bounding the second term. Because $\tilde{x}^*$ minimizes $\tilde{f}$, we have $\tilde{f}(\tilde{x}^*) \leq \tilde{f}(x^*)$, so

$$\tilde{f}(\tilde{x}^*) - f(x^*) \;\leq\; \tilde{f}(x^*) - f(x^*). \tag{3.6}$$

Moreover, since x^* minimizes f, we have $f(x^*) \leq f(\tilde{x}^*)$, so

$$f(x^*) - \tilde{f}(\tilde{x}^*) \;\leq\; f(\tilde{x}^*) - \tilde{f}(\tilde{x}^*). \tag{3.7}$$

Combining (3.6) and (3.7):

$$\left|\tilde{f}(\tilde{x}^*) - f(x^*)\right| \;\leq\; \max\left\{\tilde{f}(x^*) - f(x^*),\; f(\tilde{x}^*) - \tilde{f}(\tilde{x}^*)\right\}. \tag{3.8}$$

Combining both terms. Substituting (3.8) back into (3.5) and taking $\tilde{X} \subseteq X$ to be any feasible set with $\{x^*, \tilde{x}^*\} \subset \tilde{X}$:

$$\begin{aligned}
\left|\tilde{f}(\tilde{x}^*) - v^*\right| &\leq 2 \max_{x \in \{x^*, \tilde{x}^*\}} \left|\tilde{f}(x) - f(x)\right| && (3.9)\\
&\leq 2 \max_{x \in \tilde{X}} \left|\tilde{f}(x) - f(x)\right|. && (3.10)
\end{aligned}$$

Expanding the full and reduced objectives using the cluster partition $C_1, \ldots, C_K$ with $p_k = |C_k|/N$:

$$\begin{aligned}
\left|\tilde{f}(\tilde{x}^*) - v^*\right| &\leq 2 \max_{x \in \tilde{X}} \left| \sum_{k=1}^{K} p_k F(x, \tilde{\xi}_k) - \frac{1}{N} \sum_{i=1}^{N} F(x, \xi_i) \right| && (3.11)\\
&= 2 \max_{x \in \tilde{X}} \left| \sum_{k=1}^{K} \frac{|C_k|}{N} F(x, \tilde{\xi}_k) - \frac{1}{N} \sum_{k=1}^{K} \sum_{i \in C_k} F(x, \xi_i) \right| && (3.12)\\
&= 2 \max_{x \in \tilde{X}} \left| \sum_{k=1}^{K} \frac{|C_k|}{N} \left(F(x, \tilde{\xi}_k) - \frac{1}{|C_k|} \sum_{i \in C_k} F(x, \xi_i) \right) \right| && (3.13)\\
&\leq 2 \sum_{k=1}^{K} p_k \sup_{x \in \tilde{X}} \left| F(x, \tilde{\xi}_k) - \frac{1}{|C_k|} \sum_{i \in C_k} F(x, \xi_i) \right| && (3.14)\\
&=: 2 \sum_{k=1}^{K} p_k \, D(C_k), && (3.15)
\end{aligned}$$

where the cluster discrepancy is defined as

$$D(C_k) = \sup_{x \in \tilde{X}} \left| F(x, \tilde{\xi}_k) - \frac{1}{|C_k|} \sum_{i \in C_k} F(x, \xi_i) \right|. \tag{3.16}$$

The quantity $D(C_k)$ measures how well the representative scenario $\tilde{\xi}_k$ captures the behavior of all scenarios within cluster C_k. It quantifies the worst-case deviation, over the feasible decision set $\tilde{X}$, between the cost function evaluated at the representative scenario and the average cost over all scenarios in C_k. From a risk perspective, $D(C_k)$ represents the potential loss incurred when decisions are optimized under the representative scenario $\tilde{\xi}_k$ while any scenario in C_k may be realized. A small value of $D(C_k)$ indicates that the representative provides a good approximation for its cluster; a large value indicates poor approximation. Consequently, minimizing the weighted sum $\sum_k p_k D(C_k)$ ensures that the reduced scenario set faithfully approximates the original stochastic problem in terms of decision costs.

To obtain a practically computable bound, we restrict $\tilde{X}$ to a finite collection of optimal solutions of the one-scenario subproblems. Specifically, instead of optimizing over the full feasible set X, we use

$$\tilde{X} = \{x_1^*, x_2^*, \ldots, x_N^*\} \subset X,$$

where $x_i^* \in \arg\min_{x \in X} F(x, \xi_i)$ is the optimal first-stage decision when scenario ξ_i is realized with certainty. Each x_i^* represents the "perfect hindsight" decision for scenario ξ_i. If the minimizer is not unique, one element is selected uniformly at random.

For a fixed cluster C_k with representative $\tilde{\xi}_k$, the discrepancy (3.16) evaluated at this finite $\tilde{X}$ is approximated as:

$$D(C_k) \;\simeq\; \frac{1}{|C_k|} \left| \sum_{i \in C_k} \big(F(x_{r_k}^*, \xi_i) - F(x_{r_k}^*, \tilde{\xi}_k) \big) \right|, \tag{3.17}$$

where r_k indexes the representative scenario and $x_{r_k}^*$ is the corresponding single-scenario optimal solution. This approximation replaces the supremum over $\tilde{X}$ by the evaluation at the representative's own optimal decision, which is the most natural candidate for the worst-case solution within the cluster.

3.3 Algorithm description

The Cost-Space Scenario Clustering (CSSC) algorithm reduces a large scenario set to a smaller, more representative one while preserving the main cost characteristics of the original problem. It consists of two main steps.

Step 1: Construction of the Opportunity-Cost Matrix. In the first step, the algorithm evaluates the performance of scenario-specific optimal decisions across all scenarios. For each scenario ξ_i, the following one-scenario optimization problem is solved:

$$x_i^* \in \arg\min_{x \in X} F(x, \xi_i), \quad \forall\, i \in \{1, \ldots, N\}. \tag{3.18}$$

The optimal solution x_i^* is then evaluated under every other scenario ξ_j. The resulting values are stored in the opportunity-cost matrix $V \in \mathbb{R}^{N\times N}$:

$$V_{i,j} = F(x_i^*, \xi_j), \quad \forall\, i, j \in \{1, \ldots, N\}. \tag{3.19}$$

The entry $V_{i,j}$ is the cost incurred when the decision x_i^*, optimized for scenario ξ_i, is evaluated under scenario ξ_j instead. The diagonal entry $V_{i,i} = F(x_i^*, \xi_i)$ is the optimal cost under ξ_i itself. Two scenarios are considered cost-similar if a decision that performs well for one also performs well for the other (i.e., if the corresponding rows of V are similar). The matrix thus enables a problem-driven comparison that goes beyond geometric distance in the data space.

Algorithm 1 Step 1: Build Opportunity-Cost Matrix V

Input: Scenarios $\{\xi_1, \ldots, \xi_N\}$
Initialize $V \in \mathbb{R}^{N\times N}$ to zero
for $i = 1$ **to** N **do**
    Solve one-scenario subproblem: $x_i^* \leftarrow \arg\min_{x\in X} F(x, \xi_i)$
    Set $V_{i,i} \leftarrow F(x_i^*, \xi_i)$
    for $j = 1$ **to** N, $j \neq i$ **do**
        Fix first-stage decision: $x \leftarrow x_i^*$
        Evaluate recourse cost under ξ_j with x fixed
        Set $V_{i,j} \leftarrow F(x_i^*, \xi_j)$
        Restore x to free
    end for
end for
Output: V where $V_{i,j} = F(x_i^*, \xi_j)$

Step 2: Scenario Clustering and Representative Selection. In the second step, the N scenarios are partitioned into K clusters $C_1, \ldots, C_K$, and one representative $r_k \in C_k$ is selected for each cluster. The probability of each cluster is $p_k = |C_k|/N$. The quality of the clustering is measured by the total weighted discrepancy:

$$\sum_{k=1}^{K} p_k \left| V_{r_k,r_k} - \frac{1}{|C_k|} \sum_{j\in C_k} V_{r_k,j} \right|. \tag{3.20}$$

This discrepancy compares the diagonal entry V_{r_k,r_k} (the optimal cost of the representative under its own scenario) with the average row entry $\frac{1}{|C_k|}\sum_{j\in C_k} V_{r_k,j}$ (the average cost of the representative's decision evaluated over all scenarios in the cluster). Minimizing this measure forces each representative to provide the best average "view" of the scenarios it represents.

The joint minimization over the cluster partition and the choice of representatives can be formulated as the following mixed-integer program:

$$\min \quad \frac{1}{N} \sum_{j=1}^{N} t_j \tag{3.21}$$

$$\text{s.t.} \quad t_j \geq \sum_{i=1}^{N} x_{ij}\big(V_{j,i} - V_{j,j}\big), \qquad \forall\, j \in \{1,\dots,N\} \tag{3.22}$$

$$t_j \geq \sum_{i=1}^{N} x_{ij}\big(V_{j,j} - V_{j,i}\big), \qquad \forall\, j \in \{1,\dots,N\} \tag{3.23}$$

$$x_{ij} \leq u_j, \quad x_{jj} = u_j, \qquad \forall\, (i,j) \in \{1,\dots,N\}^2 \tag{3.24}$$

$$\sum_{j=1}^{N} x_{ij} = 1, \qquad \forall\, i \in \{1,\dots,N\} \tag{3.25}$$

$$\sum_{j=1}^{N} u_j = K \tag{3.26}$$

$$x_{ij} \in \{0,1\},\ u_j \in \{0,1\},\ t_j \geq 0, \qquad \forall\, (i,j) \in \{1,\dots,N\}^2 \tag{3.27}$$

The binary variable $u_j \in \{0,1\}$ indicates whether scenario ξ_j is selected as a cluster representative. The binary variable $x_{ij} \in \{0,1\}$ indicates whether scenario ξ_i is assigned to the cluster represented by ξ_j. The continuous variable $t_j \geq 0$ is a linearization auxiliary that equals $|V_{j,j} - \frac{1}{|C_j|}\sum_{i \in C_j} V_{j,i}|$ at optimality: constraints (3.22) and (3.23) together linearize this absolute value from above. Constraint (3.24) ensures that scenario ξ_i can only be assigned to a chosen representative, and that every representative is assigned to itself. Constraint (3.25) ensures each scenario belongs to exactly one cluster. Constraint (3.26) selects exactly K representatives.

Algorithm 2 Step 2: MIP Partitioning Problem

1: **Input:** Matrix $V \in \mathbb{R}^{N \times N}$, target size K
2: Build MIP with variables x_{ij}, $u_j \in \{0,1\}$ and $t_j \geq 0$
3: Solve MIP (3.21)–(3.27)
4: Extract representative set $\mathcal{S} \leftarrow \{j \mid u_j^* = 1\}$
5: **Output:** $\mathcal{S}$, $|\mathcal{S}| = K$

The full CSSC algorithm is summarized in Algorithm 3.

Algorithm 3 Cost-Space Scenario Clustering (CSSC)

1: **Input:** N scenarios $\{\xi_1,\dots,\xi_N\}$, probabilities $\{p_i\}$, target size K
2: Compute opportunity-cost matrix V via Algorithm 1
3: Solve MIP partitioning problem via Algorithm 2
4: **Output:** Representative set $\mathcal{S} \subseteq \{1,\dots,N\}$, $|\mathcal{S}| = K$

In this work, the CSSC algorithm is applied to the two-stage stochastic CFL problem defined in Chapter 8, the two stage stochastic UC problem in Chapter 9. The implementation within SMS++ is described in Chapter 6.

Chapter 4

Implementation in SMS++

4.1 A General Overview of SMS++

SMS++ is an open-source C++ framework for structured mathematical programming, developed at the University of Pisa. It is designed to model and solve large optimization problems that have a natural block-structured form. The system is built around a small set of core abstract classes. Each class captures a well-defined concept from mathematical optimization. Together, they form the structural backbone of every model built in SMS++.

The central concept in SMS++ is the `Block`. A `Block` represents a complete mathematical model, or a well-defined part of one. It can contain any number of `Variable` and `Constraint` objects, one `Objective`, and any number of inner `Block`s recursively. This recursive nesting allows hierarchical problem structures to be expressed naturally.

Each `Block` has two representations. The *physical representation* (PR) holds the raw instance data, such as costs, capacities, and demands. The *abstract representation* (AR) exposes the model through standard SMS++ objects: `Variable`, `Constraint`, and `Objective`. A specialized solver may work directly with the PR, while a general-purpose solver uses the AR. Both are kept synchronized automatically.

`Variable` and `Constraint` are abstract classes. The most commonly used concrete type is `ColVariable`, which represents a single real-valued scalar, possibly restricted to integer values. On the constraint side, `FRowConstraint` represents a row constraint of the form $l \leq f(x) \leq u$, where f is given by a `Function` object. Each `Variable` knows which `Block` it belongs to and can be fixed or unfixed to its current value.

The `Objective` class represents the optimization criterion of a `Block`. It can be set to minimize or maximize. In practice, `FRealObjective` is the most common concrete type, where the objective value is computed by an attached `Function`. A `LinearFunction` or `DQuadFunction` is typically used for LP and QP problems respectively.

A `Solver` is attached to a `Block` and implements a solution algorithm for it. Multiple solvers can be attached to the same `Block` simultaneously. A specialized solver works directly with the PR of a specific `Block` type, while a general-purpose solver

such as `MILPSolver` constructs the AR and passes it to an external engine like CPLEX or SCIP. When the model data changes, each solver reacts lazily: it stores incoming `Modification` objects and updates its internal state only when needed, which enables efficient re-optimization.

Whenever any part of a `Block` changes, a `Modification` object is issued to all attached solvers and to any solver attached to an ancestor `Block`. There are two kinds: physical modifications, which concern the PR and are handled by specialized solvers, and abstract modifications, which concern the AR and are handled by general-purpose solvers. This separation avoids unnecessary work. A `GroupModification` can bundle several changes together, which is useful when multiple parts of the model are updated at once.

`Configuration` objects store algorithmic parameters for `Block`s and `Solver`s. They can be serialized to and deserialized from text files, which makes it possible to configure a full solver stack without recompiling the code. In this project, solver configurations are loaded from `.txt` files such as `BSPar1_CSSC.txt`, which specifies the solver type and parameters for the CSSC algorithm.

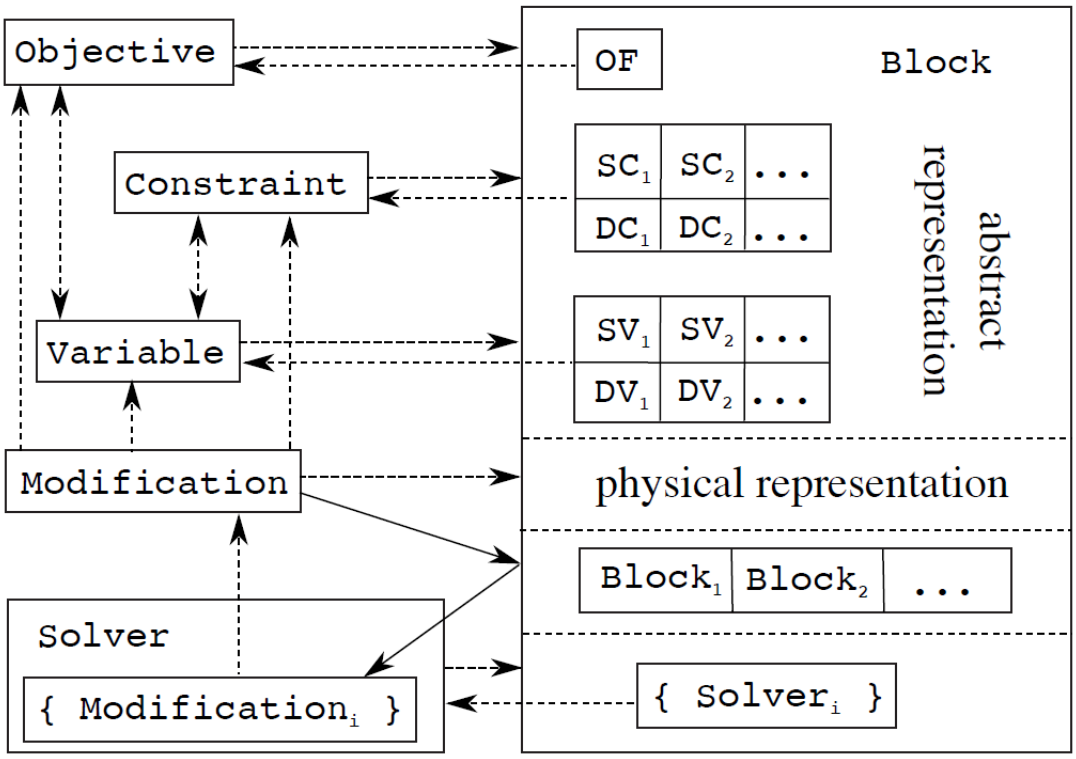

Figure 4.1: SMS++ scheme

Figure 4.1 illustrates the relationships between these components. A `Block` sits at the center, holding its PR, its AR (via `Variable`, `Constraint`, and `Objective`), its inner `Block`s, and references to its attached `Solver`s. `Modification` objects flow from the `Block` to each `Solver`, keeping all solution states consistent as the model evolves.

4.2 StochasticBlock and DiscreteScenarioSet

`StochasticBlock` is an SMS++ wrapper that makes any `Block` scenario-aware. It holds a reference to a `DiscreteScenarioSet` and uses the SMS++ methods factory to inject scenario data into the wrapped block at runtime. The block structure is not duplicated for each scenario. Instead, the same block instance is reused and only the relevant parameters are updated when the active scenario changes. This allows a solver attached to the inner block to warm-start from the previous solution, which is important for the CSSC Step 1 where N^2 sub-problems are solved sequentially.

`DiscreteScenarioSet` stores the scenario data explicitly. It holds N scenario vectors $\{\xi_1, \ldots, \xi_N\}$ and their associated probabilities $\{p_1, \ldots, p_N\}$. In the CFL context, each scenario vector encodes the client demand values under a specific realization of uncertainty. The scenario set can be loaded from a text stream or constructed programmatically. It exposes individual scenario vectors for direct access, which the CSSC Step 1 uses when injecting scenario ξ_i and ξ_j into the CFLB to compute each entry V_{ij} of the opportunity-cost matrix.

4.3 ScenarioReductionSolver

`ScenarioReductionSolver` is a specialized solver designed for scenario reduction problems formulated as CFL instances. It implements four heuristic algorithms for selecting a representative subset of K scenarios from a larger set of N scenarios. All four algorithms work by reading the pairwise transportation costs already present in the CFLB and applying a combinatorial selection strategy.

The available algorithms are: *Baseline*, a simple greedy selection based on scenario probabilities; *Dupacova*, a forward selection method that iteratively adds the scenario minimizing the Wasserstein distance (the default); *BestFit*, a local search with best-improvement selection; and *FirstFit*, a local search with first-improvement selection. The algorithm is selected via the integer parameter `intAlgorithm`, taking values 0 through 3.

A key characteristic shared by all four algorithms is that they are *data-driven*: they operate on the transportation costs in data space without solving any optimization sub-problem. They do not require an external solver and do not access the raw scenario vectors. This is the structural difference that separates them from the CSSC algorithm, which requires solving N^2 CFL sub-problems and a MIP. This difference is the direct reason why CSSC is implemented as a separate derived class rather than as a fifth algorithm inside `ScenarioReductionSolver`, as discussed in Chapter 6.

4.4 TwoStageStochasticBlock

`TwoStageStochasticBlock` (TSSB) is a generic SMS++ `Block` for modeling two-stage stochastic programming problems. It is not tied to any specific problem type: the inner `Block` can be any SMS++ block, including `CapacitatedFacilityLocationBlock`, `UCBlock`, or any other model.

Given a set of N scenarios $\{\xi_1, \ldots, \xi_N\}$ with probabilities $\{p_1, \ldots, p_N\}$ and an inner deterministic `Block` representing the per-scenario subproblem, TSSB builds the following extensive form:

$$\min \; f(x) + \sum_{k=1}^{N} p_k \, Q(x, \xi_k) \tag{4.1}$$

where x are the first-stage decisions shared across all scenarios and $Q(x, \xi_k)$ is the optimal recourse cost under scenario ξ_k.

Internally, the TSSB creates N independent copies of the inner block by serializing it once and deserializing it N times. This approach works for any `Block` type without requiring special copy methods. Scenario data is then injected into each copy using `StochasticBlock` as a temporary applicator: TSSB sets each copy as the inner block of a `StochasticBlock`, calls `set_data()` with the corresponding scenario via the `DataMappings`, and then restores the original structure.

The non-anticipativity constraints that enforce identical first-stage decisions across all N copies are built using `AbstractPath` objects. These identify the first-stage variables inside each copy by position rather than by pointer, which is necessary because the N deserialized copies live at different memory addresses. The weighted objectives are then aggregated into a single top-level objective.

Initialization requires either a `netCDF` file encoding the inner `StochasticBlock`, the `DiscreteScenarioSet`, and the `AbstractPath` objects identifying the first-stage variables, or a `ScenarioGenerator` provided programmatically via `set_scenario_generator()` after construction. Once the abstract representation is generated, any `MILPSolver` registered in SMS++ can be attached to solve the assembled problem.

4.5 CapacitatedFacilityLocationBlock

4.5.1 Block Structure

The `CapacitatedFacilityLocationBlock` (CFLB) is the SMS++ `Block` that models the CFL problem. In the standard formulation, a set of facilities must be opened to serve a set of clients at minimum total cost, subject to capacity constraints. The problem has two types of decision variables: binary variables $y_j \in \{0, 1\}$ indicating whether facility j is opened, and variables $x_{ij} \geq 0$ representing the fraction of client i's demand served by facility j.

The CFLB supports two variants of the problem. In the *unsplittable* version, each client must be served by exactly one facility. In the *splittable* version, a client's demand can be distributed across multiple facilities. It also supports three different algebraic formulations: the natural formulation (NF), the knapsack formulation (KF), and the flow formulation (FF). Each formulation exposes the same physical data through a different abstract representation, which makes the block compatible with different solver types including `LagrangianDualSolver` and `MILPSolver`.

4.5.2 Two-Stage Stochastic Structure

In the stochastic CFL context, the first-stage decisions are the facility opening variables $y_i \in \{0, 1\}$. The second-stage decisions are scenario-specific: the assignment variables $x_{ij}^k \geq 0$ are optimized independently for each scenario ξ_k once the scenario is known. The resulting extensive-form problem is:

$$\min \sum_i f_i y_i \; + \; \sum_{k=1}^{N} p_k \sum_{i,j} c_{ij} \, d_j^k \, x_{ij}^k \tag{4.2}$$

subject to demand and capacity constraints for each scenario k, and the non-anticipativity constraint that y_i is identical across all scenario copies.

The `AbstractPath` objects in this context identify the $|I|$ facility opening variables Y_i inside the `CapacitatedFacilityLocationBlock`, allowing TSSB to link them correctly across all N scenario copies.

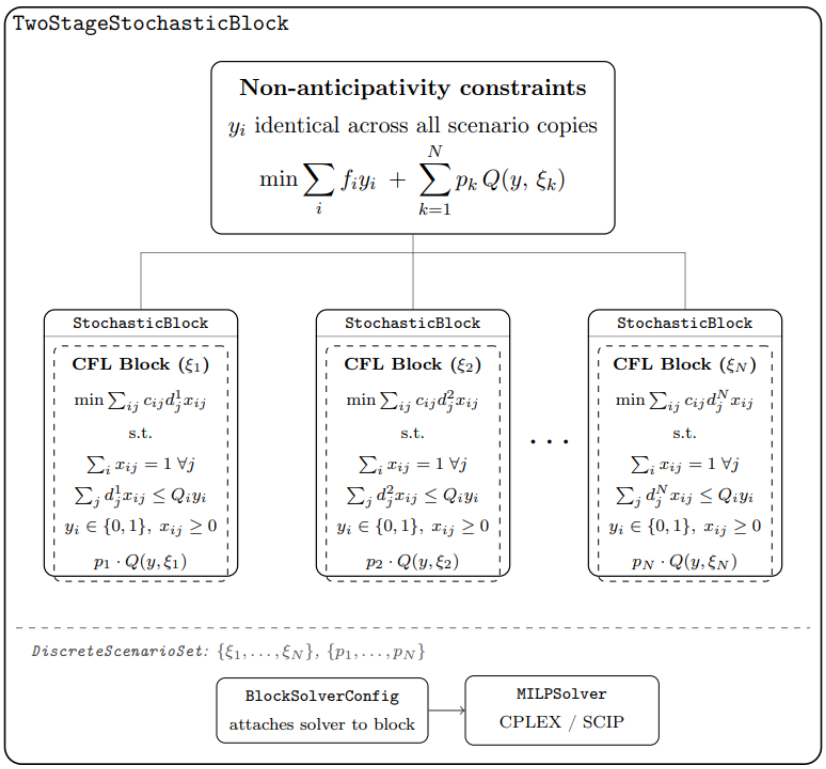

Figure 4.2: Block structure for the two-stage stochastic CFL problem in SMS++

4.5.3 Solving Pipeline

The two-stage stochastic CFL can be approached in several ways: solving the full extensive form directly, applying decomposition methods such as Lagrangian relaxation or Benders decomposition, or reducing the scenario set before solving. In this work, we focus on the scenario reduction strategy, as it is directly relevant to the CSSC algorithm developed in Chapter 6.

Phase 1: Scenario Generation

Before any stochastic problem can be built, a set of N scenarios is needed. Each scenario represents one possible realization of the uncertain parameters — in the CFL context, one possible vector of customer demands. `CFLScenarioGenerator` takes a deterministic CFL instance as input and produces N demand vectors by applying random perturbations to the base demand values. Each generated scenario is optionally validated by solving the corresponding single-scenario CFL problem, confirming that the instance remains feasible. The N scenario vectors and their associated probabilities are then stored in a `DiscreteScenarioSet` and serialized to a `netCDF` file for later use.

Phase 2: Assembling the TwoStageStochasticBlock

Once the scenarios are available, `TwoStageStochasticBlock` builds the extensive form by creating N independent copies of the inner `CapacitatedFacilityLocationBlock`, injecting one scenario into each copy, and linking the first-stage variables across all copies via non-anticipativity constraints.

The assembly proceeds as follows. The inner block is serialized once to a netCDF representation and then deserialized N times to produce N structurally identical but independent copies. For each copy k, `TwoStageStochasticBlock` temporarily wraps it inside a `StochasticBlock` and calls `set_data()` with scenario ξ_k, which applies the demand values via the block's `DataMapping` objects. The wrapper is then removed while the copy retains the injected scenario data. After all N copies are ready, the non-anticipativity constraints are added using `AbstractPath` objects, and each sub-block's objective is scaled by its scenario probability p_k before being aggregated into the combined stochastic objective.

Phase 3: Solving

With the `TwoStageStochasticBlock` fully assembled, the abstract representation is generated in the order required by SMS++:

1. `generate_abstract_variables()`, constructs all `ColVariable` objects across all sub-blocks.
2. `generate_abstract_constraints()` builds all abstract constraints across all sub-blocks, including in particular the non-anticipativity constraints linking the first-stage variables across all N scenario copies.
3. `generate_objective()`, aggregates the probability-weighted sub-block objectives into the top-level objective.

A `MILPSolver` (CPLEX, SCIP, or HiGHS) is then attached via `BlockSolverConfig::apply()` and `compute()` is called. The solver reads the full extensive form and returns the optimal first-stage solution y^* along with N recourse solutions x_k^*.

Solving with Scenario Reduction

When N is large, solving the extensive form directly becomes computationally intractable. Scenario reduction addresses this by replacing the original N scenarios with a smaller set

of $K \ll N$ representative scenarios before solving. The pipeline proceeds as follows.

Step 1: Generate scenarios. Scenarios are generated using `CFLScenarioGenerator`. It reads a deterministic CFL instance, produces N demand variations, and saves the result as a `DiscreteScenarioSet` in netCDF format. This step is performed once per instance and the output is reused across all reduction methods.

Step 2: Reduce to K representative scenarios. A `ScenarioReductionSolver` reads the `DiscreteScenarioSet` and selects K representative scenarios that best approximate the original distribution.

Step 3: Solve the reduced stochastic problem. A new `TwoStageStochasticBlock` is built using only the K representative scenarios and solved using the same procedure described above. The result is a first-stage solution $\tilde{y}^*_K$ obtained at a fraction of the computational cost of the full problem.

Table 4.1: SMS++ components involved in solving a two-stage stochastic CFL.

Component	**Phase**	**Role**
`DiscreteScenarioSet`	1, 2	Stores scenario vectors $\{\xi_k\}$ and probabilities $\{p_k\}$
`CapacitatedFacilityLocationBlock`	1, 2	Inner block encoding the CFL model for each scenario
`StochasticBlock`	2	Injects scenario data into each copy via `set_data()`
`AbstractPath`	2	Identifies first-stage variables across all N copies
`TwoStageStochasticBlock`	2	Orchestrates N copies, non-anticipativity constraints, and weighted objectives
`ScenarioReductionBlock`	2	Communication object between `DiscreteScenarioSet` and the scenario reduction solver
`ScenarioReductionSolver`	2	Selects K representative scenarios via heuristic algorithms
`BlockSolverConfig`	2, 3	Attaches a solver to the block
`MILPSolver`	3	Solves the reduced extensive form

4.6 UCBlock

4.6.1 Block Structure

`UCBlock` is the SMS++ `Block` that models the Unit Commitment (UC) problem in electrical power production. The UC problem requires finding the optimal schedule of a set of generating units over a short time horizon, minimising total operational cost while satisfying both system-wide constraints and per-unit operational constraints. `UCBlock` is designed to be flexible in its internal structure. It does not encode any specific combination of unit types or network model. Instead, it holds two abstract components.

The first is a collection of generating units, each represented by a concrete subclass of the abstract base class `UnitBlock`. Available unit types include `ThermalUnitBlock` for thermal (including nuclear) generators, `HydroUnitBlock` and `HydroSystemUnitBlock` for hydro plants, `BatteryUnitBlock` for battery storage, `IntermittentUnitBlock` for non-dispatchable renewable sources (wind, solar), and `SlackUnitBlock` as a dummy unit to absorb imbalances.

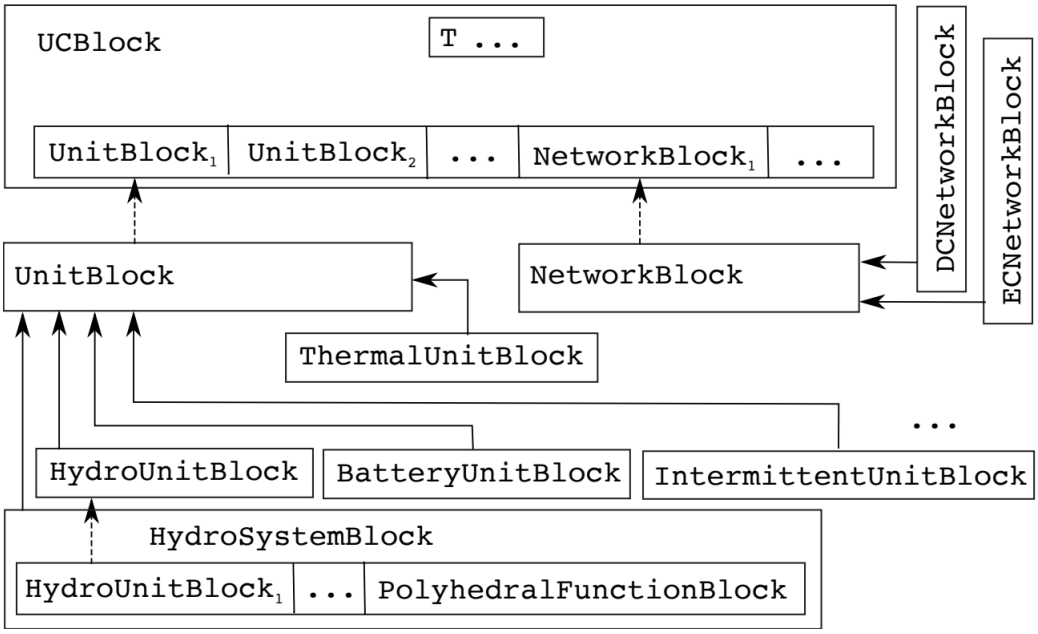

Figure 4.3: `UCBlock` scheme

The second is a network component, represented by a concrete subclass of the abstract base class `NetworkBlock`, which models how energy flows between generating units and consumption points. Available network types include `DCNetworkBlock` for the linear DC or HVDC case, `ACNetworkBlock` for the full AC case, and `ECNetworkBlock` for Energy Community instances where users share energy within a microgrid. When no network is present (the bus case), `UCBlock` handles the energy balance directly.

This modular structure means that new unit types and network models can be added independently without modifying `UCBlock` itself, and specialised solvers for specific unit types (such as `ThermalUnitDPSolver` for `ThermalUnitBlock`) can be developed and used

transparently.

4.6.2 ThermalUnitBlock

`ThermalUnitBlock` is the most important concrete subclass of `UnitBlock` for the experiments in this thesis. It models a thermal (or nuclear) generating unit with a standard set of operational constraints: minimum and maximum power output, minimum up-time and down-time, ramp-up and ramp-down limits, and start-up costs.

The commitment variable $x_{it} \in \{0, 1\}$ records the on/off state of the unit at each time period. When the unit is on ($x_{it} = 1$), the power output p_{it} must lie in $[\underline{p}_i, \overline{p}_i]$ subject to ramp constraints; when off ($x_{it} = 0$), the power output is zero. Start-up variables $v_{it} \in \{0, 1\}$ and shut-down variables $w_{it} \in \{0, 1\}$ are linked to x_{it} by the standard three-variable consistency constraints. The generation cost is a convex (often quadratic) function of p_{it} when the unit is on.

4.6.3 ECNetworkBlock

`ECNetworkBlock` is a concrete subclass of `NetworkBlock` that models the network layer of an Energy Community (EC). In an Energy Community, a set of users can buy energy from the public market, sell surplus energy back, and share energy within a microgrid. A peak tariff applies based on the maximum net power exchanged at each node over the planning horizon.

The decision variables of `ECNetworkBlock` for each node n and time interval t are: a power injection variable $P^+_{n,t} \geq 0$ (energy sold to the market), a power absorption variable $P^-_{n,t} \geq 0$ (energy bought from the market), a shared power variable $P^M_{n,t} \geq 0$ (energy traded within the microgrid in cooperative mode), and a peak power variable $P^{\max}_n \geq 0$ per node. All of these are continuous variables and belong to the second stage.

The uncertain parameter of `ECNetworkBlock` is the `ActiveDemand` matrix of shape [`NumberIntervals` × `NumberNodes`], which encodes the net demand of each user at each time step. Each scenario corresponds to one realization of this matrix. When a scenario is injected via `set_active_demand()`, only the demand values change while all prices and tariffs remain fixed.

It is important to note that `ECNetworkBlock` does not exist in isolation: it is always a component of a `UCBlock`, serving as its network layer. The generating units connected to the network (in the EC experiments, `ThermalUnitBlock` instances) produce power that is then distributed through the EC network. The combined second-stage problem includes both the production variables of the units and the network variables of `ECNetworkBlock`, so the recourse problem is not limited to the EC network variables alone.

`ECNetworkBlock` has a known serialisation bug: when `NumberIntervals` is defined explicitly in the netCDF file and differs from `NumberNodes`, the `serialize()` method reads the active demand size incorrectly. The test framework `UCScenarioReductionTest` detects this case by probing the source file and, when triggered, copies the affected netCDF group at the byte level to bypass the faulty path entirely. This is discussed further in Chapter 9.

4.6.4 Two-Stage Stochastic Structure

For the purposes of scenario reduction, `UCBlock` fits naturally into the two-stage stochastic framework. The first-stage (here-and-now) decisions are those that must be committed before the uncertain parameters are realized. The second-stage (recourse) decisions are optimized independently for each scenario once the uncertain parameters are known.

In the UC context, this separation is less straightforward than in CFL, and it is important to state it precisely. The first-stage decisions are the commitment variables $x_{it} \in \{0, 1\}$ of each thermal unit i at each time period t: these binary variables indicate whether a unit is on or off, and must be decided before the scenario is realized. In addition, depending on the problem variant, start-up and shut-down binary variables may also be first-stage.

The second-stage decisions include at minimum the power production variables $p_{it} \geq 0$ of each unit, which are continuous variables representing the power output of each unit at each time period. However, the UC second stage is in general a mixed-integer program, not a linear program. Specifically, in some UC variants, additional integer variables arise in the second stage, such as scheduling variables for managing start-up trajectories of thermal units or binary variables controlling the on/off state of battery converters. This is a key structural difference from the CFL problem, where the second stage is always a pure linear program once the first-stage facility opening variables are fixed.

The uncertain parameters in a stochastic UC formulation are typically the demand at each node and time period, and/or the renewable generation of each intermittent unit. The corresponding uncertain data objects are the `ActivePowerDemand` matrix inside the network block and the `MaxPower` profile inside each `IntermittentUnitBlock`.

4.6.5 Apply to Two-Stage Stochastic UC

The two-stage stochastic UC formulation places the commitment variables of every `ThermalUnitBlock` in the first stage and the remaining decisions (power production, network variables) in the second stage for each scenario. The resulting problem is:

$$\min \sum_{i \in \mathcal{G}} \sum_{t=1}^{T} \left[c_i^{\text{su}} v_{it} + c_i^{\text{sd}} w_{it} \right] + \sum_{\omega \in \Omega} p_\omega \, Q(x, \xi_\omega) \tag{4.3}$$

where $x = \{x_{it}, v_{it}, w_{it}\}$ are the first-stage binary variables of all thermal units, p_ω is the probability of scenario ω, and $Q(x, \xi_\omega)$ is the optimal recourse cost under scenario ξ_ω with commitment x fixed.

For each scenario ω, the recourse problem optimises:

$$Q(x, \xi_\omega) = \min \sum_{i \in \mathcal{G}} \sum_{t=1}^{T} C_i(p_{it}, x_{it}) + \sum_{n \in \mathcal{N}} \left[\pi^{\max} P_n^{\max} + \sum_{t=1}^{T} \left(\pi_t^- P_{n,t}^- - \pi_t^+ P_{n,t}^+ - \pi_t^r P_{n,t}^M \right) \right] \tag{4.4}$$

subject to per-unit operational constraints (ramp, min up/down time), power balance at each node, and network feasibility constraints, where $C_i(p_{it}, x_{it})$ is the generation cost of unit i. Here p_{it} is continuous, but as noted above, some integer variables (start-up

trajectories or converter states) may also appear in the recourse problem depending on the specific model variant. This makes the UC second stage structurally different from the CFL case where the recourse is always a pure LP.

In SMS++, `TwoStageStochasticBlock` assembles this extensive form by creating N copies of the `UCBlock`, injecting scenario ξ_ω into each copy via a `StochasticBlock` applicator with the appropriate `DataMapping`, and linking the first-stage commitment variables across all copies using `StaticAbstractPath` objects. For the EC experiments, the uncertain parameter ξ_ω corresponds to the `ActiveDemand` matrix of each copy's `ECNetworkBlock`, and the `DataMapping` maps the scenario vector directly to these entries.

4.6.6 Solving Pipeline

The two-stage stochastic UC problem follows the same three-phase pipeline as the CFL problem described in Section 4.5.3. The main difference lies in the inner block type and the nature of the uncertain parameter.

Phase 1: Scenario Generation

Scenarios for the UC problem represent different realisations of user demand. Each scenario is a vector of `ActiveDemand` values of shape [`NumberIntervals` × `NumberNodes`], encoding the net demand of each user at each time step. The N scenario vectors and their associated probabilities are stored in a `DiscreteScenarioSet` and serialized to a `netCDF` file for later use.

Phase 2: Assembling the TwoStageStochasticBlock

The assembly procedure follows the same structure as for CFL. `TwoStageStochasticBlock` creates N independent copies of the inner `UCBlock`, each of which contains an `ECNetworkBlock` as its network component. For each copy k, a `StochasticBlock` temporarily wraps the copy and calls `set_data()` with scenario ξ_k. The `DataMapping` objects registered with the applicator map the scenario vector directly to the `ActiveDemand` entries of the `ECNetworkBlock`, so that only the demand values change while all prices and tariffs remain fixed. Non-anticipativity constraints linking the first-stage commitment decisions across all N copies are then added using `AbstractPath` objects.

Phase 3: Solving

Once the `TwoStageStochasticBlock` is fully assembled, the abstract representation is generated in the same order as for CFL: `generate_abstract_variables()`, `generate_abstract_constraints()`, and `generate_objective()`. A `MILPSolver` is then attached via `BlockSolverConfig::apply()` and `compute()` is called.

Scenario reduction is applied before assembling the `TwoStageStochasticBlock`: a `ScenarioReductionSolver` reads the `DiscreteScenarioSet`, selects K representative demand scenarios, and produces a reduced `DiscreteScenarioSet` with aggregated probabilities. The

reduced problem is then assembled and solved using only K scenario copies, following the same procedure described above.

Table 4.2: SMS++ components involved in solving a two-stage stochastic UC.

Component	**Phase**	**Role**
`DiscreteScenarioSet`	1, 2	Stores scenario vectors $\{\xi_\omega\}$ and probabilities $\{p_\omega\}$
`UCBlock`	1, 2	Inner block encoding the UC model for each scenario
`ThermalUnitBlock`	1, 2	Generating unit, holds commitment variables x_{it} as first-stage decisions
`ECNetworkBlock`	1, 2	Network component, holds `ActiveDemand` as uncertain parameter
`StochasticBlock`	2	Injects scenario data into each copy via `set_data()`
`StaticAbstractPath`	2	Identifies commitment variables across all N copies
`TwoStageStochasticBlock`	2	Orchestrates N copies, non-anticipativity constraints, and weighted objectives
`ScenarioReductionBlock`	2	Communication object between `DiscreteScenarioSet` and the reduction solver
`ScenarioReductionSolver`	2	Selects K representative scenarios via heuristic algorithms
`BlockSolverConfig`	2, 3	Attaches a MILP solver to the block
`MILPSolver`	3	Solves the reduced extensive form

Chapter 5

Redesigning the Scenario Reduction Architecture in SMS++

This chapter describes the architectural redesign of the scenario reduction subsystem within the `StochasticBlock` repository of the SMS++ framework. The redesign was carried out as part of this thesis project. It constitutes the primary software engineering contribution of this work alongside the CSSC algorithm itself.

5.1 Motivation and Design Goals

5.1.1 The Problem with the Original Design

The original implementation of scenario reduction in `StochasticBlock` was built around a class called `ScenarioReductionSolver`. This class implemented several heuristic methods for scenario reduction, including Baseline, Dupacova, BestFit, and FirstFit.

The fundamental problem was that `ScenarioReductionSolver` was tightly coupled to the `Capacitated FacilityLocationBlock` (CFLB). Concretely, its `set_Block()` method performed a direct `dynamic_cast<CapacitatedFacilityLocationBlock*>` on the incoming `Block*`, and then called CFL-specific accessors such as `get_NCustomers()`, `get_Demands()`, and `get_Transportation_Costs()` directly on the resulting pointer. Because of this direct dependency, the entire `StochasticBlock` repository carried a compile-time dependency chain:

$$\texttt{StochasticBlock} \longrightarrow \texttt{CapacitatedFacilityLocationBlock}$$

This dependency was inappropriate for a general-purpose stochastic modelling library. Any user who wanted to use `StochasticBlock` for a completely unrelated problem, such as unit commitment, was forced to compile `CapacitatedFacilityLocationBlock`, `MCFBlock`, and `BinaryKnap sackBlock` as transitive dependencies, even though these components were entirely irrelevant to their use case.

A second structural problem was located in `DiscreteScenarioSet::init_representative_pool()`. This method, which is responsible for selecting a subset of K representative scenarios from

the full set of N, contained all the logic for constructing a `CapacitatedFacilityLocationBlock` internally, populating it with pairwise Wasserstein distances between scenario vectors, attaching a solver, running the optimisation, and extracting the solution from CFL-specific decision variables. In other words, a class whose single responsibility is to manage and supply scenario data was also implementing all the steps of a facility-location problem construction and solve. This made impossible to use any alternative scenario reduction algorithm, such as the CSSC method introduced in this thesis, without modifying the class itself.

5.1.2 The Core Insight Behind the Redesign

The key observation that drove the redesign was the following. `ScenarioGenerator` and `Discrete ScenarioSet` do not need to know which stochastic problem they are being used in. They only know about scenario vectors and probability weights. A scenario reduction solver likewise does not need to know anything about the outer stochastic problem; it only needs to receive the scenario data, run its algorithm, and report which scenarios were selected. The connection between the two can be mediated by a lightweight intermediary block that acts purely as a communication object. This observation, led to the introduction of `ScenarioReductionBlock`: a new class added to the `StochasticBlock` repository that holds no algorithmic logic of its own but defines a clean protocol through which any solver can receive scenario data and report its result.

5.1.3 Design Goals

The redesign was guided by three concrete goals. The first goal was to introduce a generic layer inside the `StochasticBlock` repository. This layer consists of `ScenarioReductionBlock` and a companion data structure, `ScenarioReductionBlockSolution`. Together they define the communication protocol between the scenario generator and any solver. Neither class has any dependency on `CapacitatedFacilityLocationBlock` or any other specific optimization problem. The second goal was to establish a problem-specific layer in a separate repository, called `ScenarioReductionSolver`. Rather than simply moving CFL-specific code out of `StochasticBlock`, the solver classes were rewritten to be fully generic: they read all scenario data directly from `DiscreteScenarioSet` and operate through the `ScenarioReductionBlock` interface, with no dependency on `CapacitatedFacilityLocationBlock` or any other specific block type. As a result, both `StochasticBlock` and the new `ScenarioReductionSolver` repository are self-contained. The third goal was to define a uniform interaction protocol so that the caller sets up a `ScenarioReductionBlock`, attaches any conforming solver, calls `compute()`, and retrieves the result through a single generic accessor, regardless of which algorithm was executed internally.

5.2 The Solution Data Structure

Before describing `ScenarioReductionBlock` itself, it is helpful to introduce the data structure that carries the solver's output. `ScenarioReductionBlockSolution` is a plain aggregate class defined in the same header. Its purpose is to encode the complete output

of any scenario reduction algorithm in a form that is independent of the algorithm used to produce it.

The class has three data fields and two utility methods:

ScenarioReductionBlockSolution: the complete output of any scenario reduction algorithm

```
class ScenarioReductionBlockSolution {
public:
 using ScenarioIndex = ScenarioGenerator::ScenarioIndex;

 ScenarioReductionBlockSolution() = default;

 // Indices (in the original set) of the K selected representative
    scenarios.
 std::vector< ScenarioIndex > selected_indices;

 // For each scenario i, assignments[i] is the index of the representative
 // it is assigned to. Size equals the total number of scenarios.
 std::vector< ScenarioIndex > assignments;

 // Aggregated probability weight of each representative.
 // Sums to 1.0. Same size as selected_indices.
 std::vector< double > weights;

 // Returns true if a solver has written a solution.
 [[nodiscard]] bool is_set() const {
  return( ! selected_indices.empty() );
 }

 // Resets all fields to empty.
 void clear() {
  selected_indices.clear();
  assignments.clear();
  weights.clear();
 }
};
```

The field `selected_indices` is a vector of K integers, each of which is an index into the original set of N scenarios. It identifies which scenarios were selected as representatives. The field `assignments` has exactly N entries. Entry i contains the index of the representative scenario to which scenario i was assigned. This information is needed by `DiscreteScenarioSet` to compute the aggregated probability weight of each representative: the weight of representative r is the sum of the original weights of all scenarios assigned to r. The field `weights` is the resulting vector of K aggregated probability weights, one per representative, summing to 1.0.

The method `is_set()` provides a simple predicate that the caller can use to verify that a

solver has actually written a result before attempting to read it. Because `selected_indices` is always non-empty when a valid solution exists, checking whether the vector is empty is sufficient; no additional boolean flag is needed. The method `clear()` resets all three fields to empty, which is equivalent to marking the solution as not yet set.

This design keeps the data structure entirely independent of the algorithm that produced it. A heuristic Dupacova solver and an exact MILP-based CSSC solver both write their results in exactly the same form. `DiscreteScenarioSet` reads the same fields regardless of which solver was attached.

5.3 The ScenarioReductionBlock Class

5.3.1 Role and Design Rationale

`ScenarioReductionBlock` is a subclass of the SMS++ `Block` base class designed with a deliberately minimal role. Rather than implementing scenario reduction algorithms or maintaining awareness of either the attached solver or the underlying stochastic problem type, its sole purpose is to serve as a structured communication object. It acts as a data carrier, holding pointers to the objects required by various solvers alongside a `ScenarioReductionBlockSolution` field, which the solver populates upon executing `compute()`. In practice, `DiscreteScenarioSet` instantiates a `ScenarioReductionBlock`, injects the necessary references, attaches a solver, invokes `compute()`, and subsequently retrieves the resulting solution.

This design means that adding a new scenario reduction algorithm to the framework requires only writing a new solver class in the `ScenarioReductionSolver` repository. No existing class in `StochasticBlock` needs to be modified.

5.3.2 Fields

The class holds four fields beyond those inherited from `Block`:

ScenarioReductionBlock: protected fields

```
protected:

 // Pointer to the ScenarioGenerator. Not owned by this Block.
 ScenarioGenerator * f_scenario_generator;

 // The solution written by a specialized Solver after compute().
 ScenarioReductionBlockSolution f_solution;

 // Pointer to the StochasticBlock used by CSSCScenarioReductionSolver
 // to inject scenario vectors into the inner block during Step 1.
 // Not owned.
 Block * f_stoch_applicator = nullptr;

 // Pointer to the synthetic sub-problem block (e.g. a synthetic N x N
```

```
  // CapacitatedFacilityLocationBlock) used by ScenarioReductionSolver
  // to read N, K, weights, and pairwise distances.
  // Not owned.
  Block * f_sub_problem_block = nullptr;
```

The field `f_scenario_generator` points to the `DiscreteScenarioSet` that created this block. The solver reads scenario vectors and probability weights through this pointer. Ownership is not transferred; the `DiscreteScenarioSet` outlives the `ScenarioReductionBlock` and manages its own lifetime.

The field `f_solution` is the only field held by value rather than by pointer. This is intentional: the solution is produced by the solver and lives as long as the block lives, so there is no reason to allocate it separately.

The field `f_stoch_applicator` points to a `StochasticBlock` that is used by `CSSCScenario ReductionSolver` during Step 1 of the CSSC algorithm. This applicator carries the `DataMapping` objects that know how to inject a scenario vector into the inner block's parameters (for example, customer demands for a CFL instance) without exposing the concrete block type to the solver. Not all solvers use this field, `ScenarioReductionSolver` ignores it entirely.

The field `f_sub_problem_block` points to a synthetic block, in practice a synthetic $N \times N$ `CapacitatedFacilityLocationBlock`, through which the caller communicates the values N, K, scenario weights, and pairwise distances to `ScenarioReductionSolver`. This block is never actually solved; it serves only as a structured data carrier.

In addition to these four fields, the inherited `v_Block` vector from the `Block` base class is used to hold a pointer to the `TwoStageStochasticBlock` when one is available. When `set_stochastic_block()` is called, the pointer is pushed into `v_Block[0]`. The solver then retrieves it via `get_stochastic_block()`, which returns `v_Block.front()`. This gives `CSSCScenarioReductionSolver` access to the N scenario sub-problems and to the `AbstractPath` that identifies the here-and-now variables.

5.3.3 Public Interface

The public interface provides one setter and one getter for each of the four fields:

ScenarioReductionBlock: public interface (abbreviated)

```
class ScenarioReductionBlock : public Block {
public:

 // Setters
 void set_scenario_generator( ScenarioGenerator * sg );
 void set_stochastic_block ( Block * tssb ); // stored in v_Block[0]
 void set_scenario_applicator( Block * stoch );
 void set_sub_problem_block ( Block * b );
 void set_solution( const ScenarioReductionBlockSolution & sol );
```

```
10
11  // Getters
12  ScenarioGenerator * get_scenario_generator() const;
13  Block * get_stochastic_block() const; // returns v_Block.front()
14  Block * get_scenario_applicator() const;
15  Block * get_sub_problem_block() const;
16  const ScenarioReductionBlockSolution & get_solution() const;
17 };
```

The setter `set_solution()` is called by the solver after `compute()` finishes. It copies the solver's result into `f_solution`. The getter `get_solution()` is then called by `DiscreteScenarioSet` to retrieve the result and update the pool.

5.3.4 Managing Sub-Block Ownership

One implementation detail that required careful attention was the ownership of the `TwoStageStochasticBlock` pointer placed inside `v_Block`.

In the SMS++ framework, the `Block` base class treats the contents of `v_Block` as owned sub-blocks. Normally, when a block is destroyed, its destructor iterates over `v_Block` and deletes each sub-block. This behaviour is appropriate when the sub-blocks were allocated by the parent block and have the same lifetime.

In the case of `ScenarioReductionBlock`, the situation is different. The `TwoStageStochasticBlock` that is placed in `v_Block[0]` was created by the caller and is owned by the caller. It has a longer lifetime than the `ScenarioReductionBlock`, which is created and destroyed entirely within a single call to `init_representative_pool()`. If `ScenarioReductionBlock` were to delete the `TwoStageStochasticBlock` in its destructor, the caller's `unique_ptr` would later attempt to delete the same object again, resulting in a double-free error and undefined behaviour.

The solution is to push the pointer into `v_Block` without calling `set_f_Block()` on it. In SMS++, calling `set_f_Block(this)` on a sub-block is what marks that sub-block as owned by its parent. By omitting this call, the parent pointer of the `TwoStageStochasticBlock` remains unchanged and the `ScenarioReductionBlock` destructor does not delete it. The full implementation of `set_stochastic_block()` is:

ScenarioReductionBlock::set_stochastic_block: non-owning storage of the TwoStageStochasticBlock

```
void ScenarioReductionBlock::set_stochastic_block( Block * tssb )
{
 // clear any pointer already in v_Block before replacing it
 if( ! v_Block.empty() )
  v_Block.clear();

 if( tssb )
  v_Block.push_back( tssb );
```

```
  // set_f_Block is NOT called here
  // ownership remains with the caller, this block does not delete tssb
}
```

A complementary guard is used in the test program to prevent a dangling pointer. When the `TwoStageStochasticBlock` goes out of scope, the guard clears the applicator pointer inside the `StochasticBlock` before any destructor runs:

RAII guard in the test program: clears the applicator pointer before the TwoStageStochasticBlock is destroyed

```
struct StochAppGuard {
 StochasticBlock * app;
 ~StochAppGuard() {
  if( app )
   app→set_inner_block( nullptr , false );
  }
} guard{ stoch_app };
```

This guard ensures that even if an exception is thrown before the end of the test, the applicator is always cleared before the `TwoStageStochasticBlock` is destroyed.

5.3.5 Serialization

`ScenarioReductionBlock` overrides the `serialize()` and `deserialize()` methods inherited from `Block`. The serialisation writes only the block type attribute into the netCDF group:

ScenarioReductionBlock::serialize: only the type attribute is written

```
void ScenarioReductionBlock::serialize( netCDF::NcGroup & group ) const
{
 Block::serialize( group );
 group.putAtt( "type" , "ScenarioReductionBlock" );
 // Solution is not serialized: it is produced at runtime by a Solver
}
```

The solution is intentionally not serialized. It is produced at runtime by a solver and is not part of the block's persistent description. If the application needs to store the reduction result, that is the responsibility of the caller. The `load(std::istream&)` override throws `std::logic_error` unconditionally, because text-stream loading is not meaningful for this block type.

5.4 Refactoring DiscreteScenarioSet

5.4.1 What Was Changed and Why

Before the redesign, all the logic for building a `CapacitatedFacilityLocationBlock`, populating it with Wasserstein distances, and extracting scenario assignments from CFL variables lived inside `DiscreteScenarioSet::init_representative_pool()`. The class also included three private helper methods, `create_cfl_problem_data()`, `extract_scenarios_and_assignments_from_cfl_block()`, and `generate_default_cfl_config()`, which existed solely to support this CFL-specific logic.

After the redesign, all of this CFL-specific code was removed from `DiscreteScenarioSet`. The CFL-specific logic has been removed. Scenario reduction is now delegated to a ScenarioReductionBlock and its attached Solver.

The refactored class has two new responsibilities in place of the old ones. First, it stores an externally provided `BlockSolverConfig` that specifies which solver to attach. Second, it stores an optional pointer to a `TwoStageStochasticBlock` that some solvers need. Everything else remains exactly as before: the class still loads scenarios from netCDF or from memory, still provides random pool selection via `init_random_pool()`, and still iterates over the pool via `next_scenario()`.

5.4.2 The Solver Configuration Interface

To make it possible for the caller to specify which solver to use without coupling `DiscreteScenarioSet` to any particular solver class, the class accepts a `BlockSolverConfig` object. This is the standard SMS++ mechanism for attaching a solver to a block. Two overloads are provided:

DiscreteScenarioSet: methods for providing a solver configuration

```
// transfer ownership of a BlockSolverConfig
// resets f_solution_ready so the solver will run on the next call
void set_solver_config( BlockSolverConfig * solver_config ) {
 f_solver_config.reset( solver_config );
 f_solution_ready = false;
}

// convenience overload that also sets the pool size K
void set_solver_config( BlockSolverConfig * solver_config ,
                        ScenarioIndex pool_size ) {
 f_solver_config.reset( solver_config );
 poolSize = pool_size;
 f_solution_ready = false;
}
```

Ownership of the `BlockSolverConfig` is transferred to `DiscreteScenarioSet`, which stores it in a `unique_ptr`. The flag `f_solution_ready` is reset to `false` whenever a new

configuration is provided, ensuring that the solver will be invoked again on the next call to `init_representative_pool()`.

The class also overrides `set_Block()`, which is the standard `ScenarioGenerator` method for registering a partner block. When the caller passes a `TwoStageStochasticBlock` here, `Discrete ScenarioSet` stores it in `f_stochastic_block`. This pointer is later forwarded to the `Scenario ReductionBlock` as its first sub-block, giving `CSSCScenarioReductionSolver` access to the N scenario sub-problems:

DiscreteScenarioSet::set_Block: storing the TwoStageStochasticBlock

```
void set_Block( Block * block ) override {
 f_stochastic_block = block;
}
```

5.4.3 The refactored init_representative_pool

The refactored `init_representative_pool()` method begins with several validation and early-return checks. If `target_pool_size` is `INFScenario` or equals `nbScenarios`, the full scenario set is used as the pool without invoking any solver. If `f_solution_ready` is `true`, a valid solution from a previous solver run is already cached, so the method reuses it directly without invoking the solver again. This caching behaviour is important in multi-stage workflows where `init_representative_pool()` may be called more than once after a single reduction run.

When neither early-return condition is met and a `BlockSolverConfig` has been registered, the method follows five steps:

DiscreteScenarioSet::init_representative_pool: the solver branch

```
if( f_solver_config ) {

 // Step 1: create a ScenarioReductionBlock and register this DSS
 auto srb = std::make_unique< ScenarioReductionBlock >();
 srb->set_scenario_generator( this );

 // Step 2: if a TwoStageStochasticBlock is available, add it as the first
 // sub-Block so that CSSCScenarioReductionSolver can access the
 // N sub-problems and the AbstractPath for here-and-now variables
 if( f_stochastic_block )
  srb->set_stochastic_block( f_stochastic_block );

 // Step 3: apply the solver configuration to attach a Solver to the SRB
 auto config_copy = std::unique_ptr< BlockSolverConfig >(
  f_solver_config->clone() );
 config_copy->apply( srb.get() );

 if( srb->get_registered_solvers().empty() )
```

```
    throw std::runtime_error(
     "DiscreteScenarioSet::init_representative_pool:␣"
     "no␣Solver␣registered␣after␣BlockSolverConfig::apply" );

   auto * solver = srb→get_registered_solvers().front();

   // Step 4: solve
   int status = solver→compute();
   if( status != Solver::kOK )
    throw std::runtime_error(
     "DiscreteScenarioSet::init_representative_pool:␣"
     "Solver␣failed␣with␣status␣" + std::to_string( status ) );
   solver→get_var_solution();

   // Step 5: read the solution and update the pool
   const auto & sol = srb→get_solution();
   if( ! sol.is_set() )
    throw std::runtime_error(
     "DiscreteScenarioSet::init_representative_pool:␣"
     "Solver␣did␣not␣write␣a␣solution␣into␣ScenarioReductionBlock" );

   scenarioIndexes = sol.selected_indices;
   update_pool_weights_with_assignments( sol.assignments );
   f_solution_ready = true;
  }
```

Each step serves a distinct purpose. In Step 1, a fresh `ScenarioReductionBlock` is created on the stack and the current `DiscreteScenarioSet` registers itself as the scenario generator, so the solver can access scenario vectors and weights through the block. In Step 2, the optional `TwoStageStochasticBlock` is forwarded to the block; if the solver does not need it (as is the case for `ScenarioReductionSolver`), this pointer is simply ignored. In Step 3, the `BlockSolverConfig` is cloned and applied to the block; cloning is necessary because `apply()` may consume the config object. In Step 4, `compute()` runs the solver's algorithm, and `get_var_solution()` instructs the solver to finalize and write its solution. In Step 5, the solution is read back from the block and the pool is updated.

If no `BlockSolverConfig` has been set, the method falls back to baseline selection, which simply picks the K scenarios with the highest original probability weights. This fallback ensures the class remains usable without any solver. The weight update in Step 5 is handled by the private helper method `update_pool_weights_with_assignments()`. This method iterates over all N scenarios and accumulates their original weights into the appropriate representative bin according to the assignment vector in the solution. The resulting aggregated weights are then normalized to sum to 1.0:

DiscreteScenarioSet::update_pool_weights_with_assignments: aggregating weights from the assignment vector

```
void DiscreteScenarioSet::update_pool_weights_with_assignments(
 const std::vector< ScenarioIndex > & assignments )
{
 // build a map from representative index to its position in scenarioIndexes
 std::unordered_map< ScenarioIndex , size_t > repr_to_pos;
 for( size_t pos = 0 ; pos < scenarioIndexes.size() ; ++pos )
  repr_to_pos[ scenarioIndexes[ pos ] ] = pos;

 // accumulate original weights into representative bins
 std::vector< double > agg( scenarioIndexes.size() , 0.0 );
 for( ScenarioIndex i = 0 ; i < nbScenarios ; ++i ) {
  auto it = repr_to_pos.find( assignments[ i ] );
  if( it != repr_to_pos.end() )
   agg[ it→second ] += setWeights[ i ];
  else
   throw std::runtime_error( ... );
 }

 // normalize so that weights sum to 1.0
 sumPoolWeights = std::accumulate( agg.begin() , agg.end() , 0.0 );
 poolWeights.clear();
 poolWeights.reserve( scenarioIndexes.size() );
 if( sumPoolWeights > 0.0 )
  for( const auto & w : agg )
   poolWeights.push_back( w / sumPoolWeights );
 else {
  double u = 1.0 / scenarioIndexes.size();
  poolWeights.assign( scenarioIndexes.size() , u );
 }
}
```

5.5 The Problem-Specific Layer: Two Solver Classes

All CFL-specific and CSSC-specific solver code was moved to the new `ScenarioReductionSolver` repository, which is separate from `StochasticBlock`. This repository contains two solver classes.

5.5.1 ScenarioReductionSolver

`ScenarioReductionSolver` is the refactored version of the original `ScenarioReductionSolver` developed before in the `CapacitatedFacilityLocationBlock`. It implements the four heuristic methods: Baseline, Dupacova, BestFit, and FirstFit. The internal algorithms are unchanged. The class was rewritten to be fully generic: its `set_Block()` method now casts the incoming `Block*` to `ScenarioReductionBlock*` and reads all scenario data directly

from the `DiscreteScenarioSet` held by the block. No `CapacitatedFacilityLocationBlock` is involved at any point. After `compute()` finishes, `get_var_solution()` maps the internal relative indices to absolute scenario indices, constructs a `ScenarioReductionBlockSolution`, and writes it into the block via `set_solution()`.

5.5.2 CSSCScenarioReductionSolver

`CSSCScenarioReductionSolver` implements the Cost-Space Scenario Clustering algorithm described in Chapter 6. It casts the incoming `Block*` to `ScenarioReductionBlock*` in `set_Block()` and reads N, K, and scenario data directly from the `DiscreteScenarioSet`. No synthetic sub-problem block or CFLB is required. In addition, it uses two objects specific to the CSSC algorithm: the `TwoStageStochasticBlock` retrieved via `get_stochastic_block()`, and the `StochasticBlock` applicator retrieved via `get _scenario_applicator()`. These allow the solver to inject scenario vectors into the inner block and solve the N^2 sub-problems in Step 1 without knowing the concrete block type.

5.6 The Interaction Protocol

The interaction between all components follows a fixed protocol. Steps 1–3 and 6 are common to every solver; step 4 is required only for CSSC.

1. The caller creates a `DiscreteScenarioSet` with N scenarios and loads scenario data into it.
2. The caller calls `set_solver_config()` on the `DiscreteScenarioSet` to register a `Block SolverConfig` that specifies which solver to attach and how many representatives K to select.
3. *(CSSC only)* The caller builds a `TwoStageStochasticBlock` with N sub-problems, extracts its `StochasticBlock` applicator, and calls `set_Block()` on the `Discrete ScenarioSet` to register the `TwoStageStochasticBlock`.
4. The caller calls `init_representative_pool(K)`. Internally, this creates a `Scenario Reduction Block` registers the `DiscreteScenarioSet` as the scenario generator, forwards the `TwoStage StochasticBlock` if available, applies the solver configuration, calls `compute()`, and reads the solution back.
5. The solver runs its algorithm, reads scenario data directly from the `DiscreteScenarioSet` through the `ScenarioReductionBlock`, and writes its result via `set_solution()`.
6. The caller reads the pool via `get_selected_scenarios()` and constructs a reduced `Discrete ScenarioSet` with K scenarios and aggregated weights. The reduced stochastic problem can now be assembled and solved with no further knowledge of which reduction algorithm was used.

From the caller's point of view, steps 4–5 are entirely transparent. The caller only needs to call `set_solver_config()` once and then invoke `init_representative_pool(K)`. The `ScenarioReductionBlock` is created and destroyed entirely within that single method call.

5.7 Repository Structure After the Redesign

Before the redesign, the dependency graph was:

`StochasticBlock` $\longrightarrow$ `CapacitatedFacilityLocationBlock` $\longrightarrow$ `MCFBlock`, `BinaryKnapsackBlock`.

After the redesign, the transitive dependency is removed from `StochasticBlock`:

`StochasticBlock` (no dependency on CFLB or any other problem)

`ScenarioReductionSolver` $\longrightarrow$ `StochasticBlock`, `CapacitatedFacilityLocationBlock`.

Table 5.1 summarizes all files created, moved, or removed by the redesign.

Table 5.1: Files created, moved, or removed by the redesign.

File	Status	Change
`StochasticBlock/include/ScenarioReductionBlock.h`	New	Generic block and solution struct; no CFLB dependency
`StochasticBlock/src/ScenarioReductionBlock.cpp`	New	`set_stochastic_block`, `print`, `serialize`, `deserialize`
`StochasticBlock/include/DiscreteScenarioSet.h`	Modified	Removed CFL helpers; added `set_solver_config`, `set_Block`, `f_solver_config`, `f_stochastic_block`, `f_solution_ready`
`StochasticBlock/src/DiscreteScenarioSet.cpp`	Modified	Removed `create_cfl_problem_data`, `generate_default_cfl_config`; refactored `init_representative_pool`
`ScenarioReductionSolver/include/ScenarioReductionSolver.h`	New (moved, rewritten)	Generic heuristic solver; reads directly from DiscreteScenarioSet
`ScenarioReductionSolver/src/ScenarioReductionSolver.cpp`	New (moved, rewritten)	`set_Block` reads from DSS; `get_var_solution` writes solution
`ScenarioReductionSolver/include/CSSCScenarioReductionSolver.h`	New	CSSC solver; requires TSSB and applicator
`ScenarioReductionSolver/src/CSSCScenarioReductionSolver.cpp`	New	Full CSSC pipeline (Step 1 V-matrix and Step 2 MILP)

5.8 Comparison with the Previous Design

Table 5.2 contrasts the key properties of the previous design with the redesigned architecture.

Table 5.2: Previous design compared with the redesigned architecture.

Property	**Previous design**	**Redesigned**
Block type required by solver	`CapacitatedFacility-LocationBlock` (direct cast in `set_Block`)	`ScenarioReductionBlock` (generic; data read via accessors)
`StochasticBlock` dependency on CFLB	Yes (compile-time transitive)	No (CFLB confined to `ScenarioReductionSolver` repo)
CFL logic in `DiscreteScenarioSet`	Yes (CFLB constructed inside `init_representative_pool`)	No (solver attached externally via `set_solver_config`)
Solution representation	CFL decision variables y/x (problem-specific)	`ScenarioReductionBlockSolution` (generic indices and weights)
Adding a new reduction algorithm	Required modifying `ScenarioReductionSolver` directly	New solver subclass in its own repo; no existing code touched
Repository of solver classes	`StochasticBlock`	Separate `ScenarioReductionSolver` repo (both solvers generic)

The most important consequence of the redesign is that `StochasticBlock` is now entirely self-contained. A user of the stochastic modelling infrastructure who has no interest in facility-location problems no longer needs to compile `CapacitatedFacilityLocationBlock`, `MCFBlock`, or `BinaryKnapsackBlock`. Extending the framework to a new problem type only requires adding a new solver class in the `ScenarioReductionSolver` repository, with no changes to `StochasticBlock` or any of its existing classes.

The two new classes, `ScenarioReductionBlock` and `ScenarioReductionBlockSolution`, did not exist before this work. Their introduction, together with the refactoring of `DiscreteScenarioSet` and the relocation of the solver layer into a separate repository, constitutes the primary software engineering contribution of this thesis alongside the CSSC algorithm itself.

Chapter 6

Implementation of the Scenario Reduction Solvers

Chapter 5 described the two-layer architecture introduced by the redesign: a generic `Scenario ReductionBlock` in the `StochasticBlock` repository, and a separate `Scenario ReductionSolver` repository housing the problem-specific solver classes. This chapter describes the implementation of both solver classes in that repository.

The work involved two distinct tasks. The first was a *refactoring* of the existing `Scenario ReductionSolver`, which was tightly coupled to `CapacitatedFacilityLocationBlock` (CFLB), into a fully generic class that works directly from a `DiscreteScenarioSet` with no dependency on any specific block type. The four heuristic algorithms (Baseline, Dupacova, BestFit, FirstFit) are preserved without modification, only the data-loading layer changed. The second task was the *creation* of a new class, `CSSCScenarioReductionSolver`, which implements the Cost-Space Scenario Clustering algorithm. This chapter documents the final, generic version.

6.1 Refactoring ScenarioReductionSolver

6.1.1 What Changed and Why

The original `ScenarioReductionSolver` was developed and in the `CapacitatedFacilityLocationBlock` repository. Its `set_Block()` method received a `Block*` and immediately cast it to `CapacitatedFacilityLocationBlock*`, then called CFL-specific methods to read the problem data:

Original set_Block(): direct cast to CFLB (old design)

```
1 void ScenarioReductionSolver::set_Block( Block * block ) {
2  auto * cfl = dynamic_cast< CapacitatedFacilityLocationBlock * >( block );
3  if( ! cfl )
4   throw std::invalid_argument( "block must be a CFLB" );
5  // read N, K, weights, distances from CFL-specific accessors
```

```
 refresh_cached_data( *cfl , cfl→get_NMaxFacilities() );
}
```

Inside `refresh_cached_data()`, the solver called `get_NCustomers()`, `get_Demands()`, and `get _Transportation_Costs()` directly on the CFLB pointer. This meant that to use the heuristics, the caller had to build a synthetic $N \times N$ CFLB just to encode the scenario weights and pairwise distances, even when the actual stochastic problem had nothing to do with facility location.

The refactored class reads all of this data directly from the `DiscreteScenarioSet` held by the `ScenarioReductionBlock`:

New set_Block(): reads directly from DiscreteScenarioSet

```
void ScenarioReductionSolver::set_Block( Block * block ) {
 Solver::set_Block( block );
 if( ! block ) return;

 auto * srb = dynamic_cast< ScenarioReductionBlock * >( block );
 if( ! srb )
  throw std::invalid_argument(
   "ScenarioReductionSolver::set_Block: block must be a "
   "ScenarioReductionBlock" );

 auto * dss = dynamic_cast< DiscreteScenarioSet * >(
  srb→get_scenario_generator() );
 if( ! dss )
  throw std::invalid_argument(
   "ScenarioReductionSolver::set_Block: ScenarioGenerator is not a "
   "DiscreteScenarioSet" );

 nb_atoms = static_cast< Index >( dss→get_poolSize() );
 const auto sc_dim = dss→get_scenarioSize();

 // pool_map: relative [0..N-1] to absolute DSS scenario index
 const auto pool = dss→get_selected_scenarios();
 f_pool_map.assign( pool.begin() , pool.end() );

 // uniform weights 1/N
 f_weights.assign( nb_atoms , nb_atoms ? 1.0 / nb_atoms : 0.0 );

 // pairwise Euclidean distance matrix (Wasserstein ground metric)
 f_dist.assign( nb_atoms ,
  std::vector< double >( nb_atoms , 0.0 ) );
 for( Index i = 0 ; i < nb_atoms ; ++i )
  for( Index j = i + 1 ; j < nb_atoms ; ++j ) {
   double d2 = 0.0;
   for( std::size_t d = 0 ; d < sc_dim ; ++d ) {
```

```
35      const double dd =
36       dss→get_scenario_value( f_pool_map[i] , d ) -
37       dss→get_scenario_value( f_pool_map[j] , d );
38      d2 += dd * dd;
39     }
40     const double dist = std::sqrt( d2 );
41     f_dist[i][j] = dist;
42     f_dist[j][i] = dist;
43    }
44
45   reduced_atoms.assign( nb_atoms , false );
46   ind_red.clear();
47   f_solution_value = 0.0;
48  }
```

Three observations are worth making. First, the pairwise Euclidean distance matrix is computed directly from the scenario feature vectors obtained via `get_scenario_value()`, so no synthetic CFLB is needed at all. Second, uniform weights $1/N$ are assigned because the solver has no information about non-uniform weights from the `DiscreteScenarioSet` pool, if the pool was initialised with non-uniform weights, the caller is responsible for ensuring that the pool reflects them. Third, the pool map `f_pool_map` translates relative indices $0, \ldots, N-1$ over the active pool to absolute scenario indices in the full `DiscreteScenarioSet`; this translation is needed when writing back the solution.

6.1.2 What Did Not Change

The four heuristic algorithms themselves, Baseline, Dupacova, BestFit, and FirstFit, are unchanged. They operate on the internal fields `nb_atoms`, `nb_reduced`, `f_weights`, and `f_dist`, and produce output through `ind_red`, `reduced_atoms`, and `f_solution_value`. The only difference is that these fields are now populated from the `DiscreteScenarioSet` rather than from a CFLB.

6.1.3 Writing the Solution Back

After `compute()` finishes, `get_var_solution()` maps the relative representative indices back to absolute DSS indices and constructs a `ScenarioReductionBlockSolution`:

get_var_solution(): writing the result into ScenarioReductionBlock

```
1  void ScenarioReductionSolver::get_var_solution(
2   Configuration * solc )
3  {
4   auto * srb = dynamic_cast< ScenarioReductionBlock * >( get_Block() );
5
6   ScenarioReductionBlockSolution sol;
7
8   // map relative representative indices to absolute pool indices
```

```
 sol.selected_indices.reserve( ind_red.size() );
 for( auto r : ind_red )
  sol.selected_indices.push_back( f_pool_map[ r ] );

 // aggregate weights: each scenario assigns its weight to its
 // nearest representative
 sol.assignments.resize( nb_atoms );
 sol.weights.assign( ind_red.size() , 0.0 );
 for( Index i = 0 ; i < nb_atoms ; ++i ) {
  // find nearest representative
  Index best_rel = ind_red[ 0 ];
  double best_c = f_dist[ i ][ best_rel ];
  for( auto r : ind_red )
   if( f_dist[i][r] < best_c )
    { best_c = f_dist[i][r]; best_rel = r; }
  sol.assignments[i] = f_pool_map[ best_rel ];
  // accumulate weight
  auto it = std::find( sol.selected_indices.begin() ,
                       sol.selected_indices.end() ,
                       sol.assignments[i] );
  if( it != sol.selected_indices.end() )
   sol.weights[ std::distance(
    sol.selected_indices.begin() , it ) ] += f_weights[i];
 }

 srb->set_solution( sol );
}
```

An important difference from the old design is that assignments are now computed by nearest-neighbour in the distance matrix rather than read from the CFLB x variables. This is correct because the heuristic algorithms minimize the Wasserstein distance, which is exactly the sum of minimum distances to representatives.

6.1.4 Comparison Between Old and New Design

Table 6.1 summarises the key differences between the original class and the refactored one.

Table 6.1: ScenarioReductionSolver: original vs. refactored design.

Property	**Original**	**Refactored**
Repository	`CapacitatedFacility-LocationBlock`	`ScenarioReductionSolver`
Block type required	`CapacitatedFacility-LocationBlock` (direct cast)	`ScenarioReductionBlock` with a `DiscreteScenarioSet`
Synthetic CFLB needed	Yes (caller must build it to pass N, K, weights, distances)	No
Distance matrix	Read from CFLB transportation costs	Computed from scenario feature vectors (Euclidean)
Problem scope	CFL only	Any two-stage stochastic problem
Solution output	Written to CFLB y/x variables	Written to `ScenarioReductionBlockSolution`
Core algorithms	Baseline, Dupacova, BestFit, FirstFit	Same (unchanged)

6.2 CSSCScenarioReductionSolver

The `CSSCScenarioReductionSolver` was not part of the original codebase. It is created as a new class to implement the Cost-Space Scenario Clustering algorithm, which is one of the main algorithmic contributions of this thesis.

6.2.1 Class Structure

`CSSCScenarioReductionSolver` inherits directly from `Solver` and is self-contained. Its class declaration is:

CSSCScenarioReductionSolver: class declaration (abbreviated)

```
class CSSCScenarioReductionSolver : public Solver {
public:
 // callable types for the two extensibility hooks
 using VarExtractor = std::function<
  std::vector< ColVariable * >( Block * ) >;
 using DataInjector = std::function<
  void( Block * , const std::vector< double > & ) >;

 void set_Block( Block * block ) override;
 int compute( bool changedvars = false ) override;
 void get_var_solution( Configuration * solc = nullptr ) override;
```

```
12
13   void set_milp_config( BlockSolverConfig * cfg );
14   void set_nb_reduced( int K );
15   void set_var_extractor( VarExtractor extractor );
16   void set_data_injector( DataInjector injector );
17   void set_fix_with_modification( bool fwm );
18   void set_milp_time_limit( double seconds );
19  private:
20   TwoStageStochasticBlock * f_tssb = nullptr;
21   StochasticBlock * f_stoch_applicator = nullptr;
22   BlockSolverConfig * f_milp_config = nullptr;
23   VarExtractor f_var_extractor;
24   DataInjector f_data_injector;
25   bool f_fix_with_mod = true;
26   double f_milp_time_limit = 120.0;
27   int f_N = 0;
28   int f_K = 0;
29   std::vector< Index > f_pool_map;
30   std::vector< double > f_weights;
31   std::vector< std::vector< double > > f_dist;
32   ...
33  };
```

The class stays problem-agnostic through one key design choice. The computation delegates to caller-provided callbacks and a configuration flag. It does not hard-code any block-type-specific logic.

The first hook is the `VarExtractor`. It is also the only strictly required hook. `generate_abstract_variables()` runs first. The extractor then receives the inner block. It returns pointers to all first-stage, here-and-now, `ColVariable` objects. For CFL these are the y_f binary variables: they say whether facility f is open. For unit commitment they are the commitment variables x_{it} of every `ThermalUnitBlock`. The solver uses these pointers to read y_i^* after the diagonal sub-problem is solved. It also uses them to fix and release the variables during the off-diagonal solves. If the extractor returns an empty list, `compute_generic_V_matrix()` throws right away. No solver has been attached yet at that point.

The second hook is the `DataInjector`. A boolean flag controls whether it is used at all: `f_fix_with_mod`, set via `set_fix_with_modification()`, default `true`. Both branches below reuse the same persistent inner block. They also reuse the same attached solver, for the entire N^2 loop. This part of the algorithm never constructs a fresh `Block`. The two branches differ in one thing only: how the solver learns that new scenario data has been written into the block.

inject_scenario(): the two real scenario-injection paths

```
1  auto inject_scenario = [ & ]( const std::vector< double > & data ) {
2   if( f_fix_with_mod ) {
```

```
   f_stoch_applicator→set_data( data , eNoBlck , eNoBlck );
   if( f_data_injector ) f_data_injector( inner , data );
  } else {
   f_stoch_applicator→set_data( data , eModBlck , eModBlck );
   inner→add_Modification( std::make_shared< NBModification >( inner ) );
  }
};
```

`set_Block()` validates the incoming block. It must be a `ScenarioReductionBlock`. The required objects must already be registered with it. `set_Block()` then reads the pool size. It builds the pool map. It sets uniform weights. N comes from the `DiscreteScenarioSet`. K is set separately, via `set_nb_reduced()`.

set_Block(): validation and data extraction

```
void CSSCScenarioReductionSolver::set_Block( Block * block )
{
 std::lock_guard< std::recursive_mutex > lock( f_mutex );
 if( block == f_Block ) return;

 Solver::set_Block( block );
 if( ! f_Block ) return;

 auto * srb = dynamic_cast< ScenarioReductionBlock * >( f_Block );
 if( ! srb )
  throw std::invalid_argument( "block␣must␣be␣a␣ScenarioReductionBlock" );

 // CSSC requires a TwoStageStochasticBlock for Step 1
 f_tssb = dynamic_cast< TwoStageStochasticBlock * >(
  srb→get_stochastic_block() );
 if( ! f_tssb )
  throw std::invalid_argument( "ScenarioReductionBlock␣must␣contain␣"
   "a␣TwoStageStochasticBlock" );

 // and a StochasticBlock applicator to inject scenario data
 f_stoch_applicator = dynamic_cast< StochasticBlock * >(
  srb→get_scenario_applicator() );
 if( ! f_stoch_applicator )
  throw std::invalid_argument( "ScenarioReductionBlock␣must␣contain␣"
   "a␣StochasticBlock␣applicator" );

 auto * dss = dynamic_cast< DiscreteScenarioSet * >(
  srb→get_scenario_generator() );
 if( ! dss )
  throw std::invalid_argument( "ScenarioGenerator␣must␣be␣a␣"
   "DiscreteScenarioSet" );

 const int N = static_cast< int >( dss→get_poolSize() );
```

```
34  const auto pool = dss→get_selected_scenarios();
35  f_pool_map.assign( pool.begin() , pool.end() );
36  f_N = N;
37
38  // uniform weights
39  f_weights.assign( N , 1.0 / static_cast< double >( N ) );
```

`compute()` validates that all required objects are present and then orchestrates the two-step pipeline:

compute(): validation and two-step pipeline

```
1  int CSSCScenarioReductionSolver::compute( bool )
2  {
3   std::lock_guard< std::recursive_mutex > lock( f_mutex );
4   if( ! get_Block() ) return kError;
5
6   if( f_K <= 0 || f_K > f_N )
7    throw std::logic_error( "call␣set_nb_reduced(K)␣with␣0␣<␣K␣<=␣N" );
8   if( ! f_milp_config )
9    throw std::logic_error( "call␣set_milp_config()␣before␣compute()" );
10  if( ! f_var_extractor )
11   throw std::logic_error( "call␣set_var_extractor()␣before␣compute()" );
12
13  mod_clear();
14
15  // Step 1: build the N x N opportunity-cost matrix V
16  f_V_matrix = compute_generic_V_matrix();
17
18  // Step 2: solve the CSSC partitioning MILP using V
19  solve_cssc_milp( f_V_matrix );
20
21  return kOK;
22 }
```

6.2.2 Step 1: Building the Opportunity-Cost Matrix

$V[i][j]$ is the total cost incurred when the system is designed for scenario ξ_i (the first-stage decisions y_i^* are optimal for ξ_i) but must serve scenario ξ_j. Formally:

$$V[i][j] = F(y_i^*, \xi_j)$$

where $F(y, \xi)$ is the total cost of serving scenario ξ with first-stage decisions y fixed. The diagonal $V[i][i]$ is the optimal cost when design and realization coincide. This matrix called opportunity-cost matrix of representing scenario j by scenario i: how much more expensive does it become to serve j.

The function `compute_generic_V_matrix()` fills the matrix with a single strategy. It uses whichever of the two injection paths from Section 6.2 is selected by `f_fix_with_mod`. A

single inner block is obtained from the applicator. It stays alive for the entire N^2 loop, no matter which path is active. Before the loop starts, abstract variables, constraints, and the objective are generated once. A solver is attached once and stays attached. For each row i, scenario i is injected via `inject_scenario()`. The diagonal MIP is then solved. If this diagonal solve does not reach `kOK` or `kLowPrecision`, the function throws right away. The row is abandoned at that point (see below). Otherwise the first-stage values y_i^* are read. The variables are relaxed to continuous and fixed at y_i^*. Then, for each column j, scenario j is injected. The LP is then solved to obtain $V[i][j]$. After all columns are done, the variables are unfixed and restored to binary for the next row.

Step 1 loop skeleton (shared by both injection paths)

```
Block * inner = f_stoch_applicator→get_inner_block();
inner→generate_abstract_variables();
inner→generate_abstract_constraints();
inner→generate_objective();
sub_cfg→apply( inner );
Solver * sub_solver = inner→get_registered_solvers().front();

for( int i = 0 ; i < n ; ++i ) {
 // (a) inject xi, solve MIP, read y*_i and V[i][i]
 inject_scenario( data_i );
 sub_solver→compute();
 sub_solver→get_var_solution();
 V[i][i] = sub_solver→get_var_value();
 for each y in fs_vars: y_star[k] = y→get_value();

 // (b) relax binary →continuous, fix at y*_i
 for each y: y→set_type( relax_type , eNoMod );
 for each y: y→set_value( y_star[k] );
 for each y: y→is_fixed( true , fix_flag );

 // (c) for each j != i: inject xj, solve LP, read V[i][j]
 for( int j = 0 ; j < n ; ++j ) {
  if( j == i ) continue;
  inject_scenario( data_j );
  sub_solver→compute();
  sub_solver→get_var_solution();
  V[i][j] = sub_solver→get_var_value();
 }

 // (d) unfix and restore binary for next row's MIP
 for each y: y→is_fixed( false , fix_flag );
 for each y: y→set_type( orig_type , eNoMod );
}
sub_cfg→clear(); delete sub_cfg;
```

If a sub-problem solve returns a status other than `kOK` or `kLowPrecision`, the entry is

set to the sentinel value -1.0. In the Step 2 MILP, any x_{ij} corresponding to an infeasible entry is forced to zero by an explicit constraint, ensuring that the partitioning never assigns scenario i to representative j when serving j with design y_i^* is infeasible.

6.2.3 Step 2: The CSSC Partitioning MILP

The partitioning MILP used in Step 2 is formulated and analyzed in full in Chapter 3, Given the opportunity-cost matrix V, the problem selects K representatives $u_j \in \{0, 1\}$, assigns every scenario via $x_{ij} \in \{0, 1\}$, and minimizes the weighted clustering discrepancy via auxiliary variables $t_j \geq 0$ (equations (3.21)–(3.27)). The present section describes how this formulation is constructed and solved inside SMS++.

The MILP is constructed in-memory using `AbstractBlock`, the generic SMS++ block type that accepts manually added variables, constraints, and an objective:

solve_cssc_milp(): variable and constraint construction

```
auto milp_block = std::make_unique< AbstractBlock >();

// heap allocation: AbstractBlock takes ownership and deletes in destructor
auto * x_vars_p = new std::vector< ColVariable >( n * n );
auto * u_vars_p = new std::vector< ColVariable >( n );
auto * t_vars_p = new std::vector< ColVariable >( n );
auto & x_vars = *x_vars_p;
auto & u_vars = *u_vars_p;
auto & t_vars = *t_vars_p;

for( int i = 0 ; i < n * n ; ++i )
 x_vars[i].set_type( ColVariable::kBinary );
for( int j = 0 ; j < n ; ++j )
 u_vars[j].set_type( ColVariable::kBinary );
for( int j = 0 ; j < n ; ++j )
 t_vars[j].set_type( ColVariable::kContinuous );

milp_block→add_static_variable( x_vars );
milp_block→add_static_variable( u_vars );
milp_block→add_static_variable( t_vars );
```

Variables must be heap-allocated because `AbstractBlock` takes ownership of all registered variable containers and calls `delete` on them in its destructor. Stack-allocated vectors would cause a double-free fault.

Infeasible pairs (sentinel $V[j][i] < 0$) are excluded by adding explicit constraints $x_{ij} \leq 0$ before the main constraint families. The absolute-value constraints are then added for each j:

solve_cssc_milp(): absolute-value constraints

```
for( int j = 0 ; j < n ; ++j ) {
 // eq.(3.22): t_j - S_j >= 0 (t_j >= S_j)
 LinearFunction::v_coeff_pair pos;
 pos.emplace_back( &t_vars[j] , 1.0 );
 for( int i = 0 ; i < n ; ++i ) {
  if( V[j][i] < 0.0 ) continue; // skip infeasible
  const double c = V[j][i] - V[j][j];
  if( std::abs(c) > 1e-15 )
   pos.emplace_back( &x_vars[i*n+j] , -c );
 }
 // add FRowConstraint lhs=0.0, rhs=+inf

 // eq.(3.23): t_j + S_j >= 0 (t_j >= -S_j)
 LinearFunction::v_coeff_pair neg;
 neg.emplace_back( &t_vars[j] , 1.0 );
 for( int i = 0 ; i < n ; ++i ) {
  if( V[j][i] < 0.0 ) continue;
  const double c = V[j][i] - V[j][j];
  if( std::abs(c) > 1e-15 )
   neg.emplace_back( &x_vars[i*n+j] , c );
 }
 // add FRowConstraint lhs=0.0, rhs=+inf
}
```

The objective $\frac{1}{N}\sum_j t_j$ is added as an `FRealObjective` with sense `eMin`. The MILP is solver-agnostic: any MILP solver registered in SMS++ (CPLEX, Gurobi, HiGHS, SCIP) can be used without any code change.

A time limit is applied to the Step 2 MILP. This only happens if `f_milp_time_limit` is positive. It happens via `set_par(Solver::dblMaxTime, f_milp_time_limit)`. A non-positive value leaves the solver unbounded. This is what happens, for example, if the caller never calls `set_milp_time_limit()`. The solver can stop because of that time limit. It can also stop because of an iteration limit. Either way, the result is accepted only if a feasible incumbent was actually found: its objective value must be finite. When that condition holds, a warning is printed. The best incumbent is then used as a valid clustering, even if it may be suboptimal. If the solver stops at the limit with no feasible incumbent at all, `solve_cssc_milp()` throws instead. It does not return a meaningless result.

After the MILP is solved, representatives are identified by reading the u_j variables with threshold 0.5. The result is stored in `ind_red`. This vector is used right below to extract the assignments. The result is also copied into a separate field, `f_representatives`. `get_var_solution()` reads from this field, not from `ind_red`. The dedicated copy avoids ambiguity: `ind_red` is also written and read by the heuristic `ScenarioReductionSolver` code path from Section 6.1, and it gets reset by `set_Block()`.

solve_cssc_milp(): extracting representatives and assignments

```
// extract u_j →representatives
ind_red.clear();
for( int j = 0 ; j < n ; ++j )
 if( u_vars[j].get_value() > 0.5 )
  ind_red.push_back( static_cast< Index >(j) );
f_representatives = ind_red;

// fallback: if MILP returned wrong count due to numerical tolerance,
// rank by u_j value and take top K
if( static_cast< int >( ind_red.size() ) != K ) {
 std::vector< int > all(n);
 std::iota( all.begin() , all.end() , 0 );
 std::sort( all.begin() , all.end() , [&]( int a , int b ){
  return u_vars[a].get_value() > u_vars[b].get_value();
 });
 ind_red.clear();
 for( int k = 0 ; k < K ; ++k )
  ind_red.push_back( static_cast< Index >( all[k] ) );
 std::sort( ind_red.begin() , ind_red.end() );
 f_representatives = ind_red;
}

// extract x_ij →assignments
f_assignments.assign( n , ind_red[0] );
for( int i = 0 ; i < n ; ++i )
 for( int j = 0 ; j < n ; ++j )
  if( x_vars[i*n+j].get_value() > 0.5 )
   { f_assignments[i] = static_cast< Index >(j); break; }
```

The fallback ranking handles one case: the solver returns slightly infeasible binary values. After the assignments are extracted, three things happen. `update_reduced_atoms()` marks the selected scenarios in the `reduced_atoms` boolean vector. `indices_to_choose` is populated with the complementary set, the non-representative scenario indices. The Wasserstein distance is computed from the pairwise Euclidean distance matrix `f_dist`, for reporting purposes.

`get_var_solution()` maps relative indices back to absolute DSS indices and writes the `Scenario ReductionBlockSolution`:

get_var_solution(): constructing the ScenarioReductionBlockSolution

```
void CSSCScenarioReductionSolver::get_var_solution( Configuration * )
{
 auto * srb = dynamic_cast< ScenarioReductionBlock * >( f_Block );

 ScenarioReductionBlockSolution sol;
```

```
6
7   // map relative →absolute indices
8   sol.selected_indices.reserve( f_representatives.size() );
9   for( auto r : f_representatives )
10   sol.selected_indices.push_back( f_pool_map[ r ] );
11
12  // assignments
13  sol.assignments.resize( static_cast< std::size_t >( f_N ) );
14  for( int i = 0 ; i < f_N ; ++i )
15   sol.assignments[ i ] = f_pool_map[ f_assignments[ i ] ];
16
17  // aggregate uniform weights by assignment
18  sol.weights.resize( f_representatives.size() , 0.0 );
19  for( int i = 0 ; i < f_N ; ++i ) {
20   const Index rep_abs = sol.assignments[ i ];
21   auto it = std::find( sol.selected_indices.begin() ,
22                        sol.selected_indices.end() , rep_abs );
23   if( it != sol.selected_indices.end() )
24    sol.weights[ std::distance(
25     sol.selected_indices.begin() , it ) ] += f_weights[ i ];
26  }
27
28  srb→set_solution( sol );
29 }
```

6.3 BlockSolverConfig Lifecycle

A `BlockSolverConfig` manages solver attachment to a block. Calling `apply(block)` registers the solver, calling `clear()` unregisters it. The `clear()` call is mandatory before the config is released: without it, the block retains a dangling pointer to the destroyed solver.

`CSSCScenarioReductionSolver` uses the same `f_milp_config` for both Step 1 sub-problem solves and the Step 2 `AbstractBlock`. It always works with a clone, though, never the original. The solver itself owns the original object passed to `set_milp_config()`. It deletes this object only in its own destructor (`~CSSCScenarioReductionSolver(){ delete f_milp_config; }`). So callers transfer ownership when they call `set_milp_config()`.

In `compute_generic_V_matrix()`, the Step 1 clone is applied to the inner block. It is cleared and deleted after the N^2 loop completes. In `solve_cssc_milp()`, the Step 2 clone is applied to the `AbstractBlock`. It is cleared and deleted right after the solution is read. This cleanup runs correctly on the normal path. It also runs correctly on one specific failure path: when no solver could be attached to the inner block. But it does not run on every exit path. Suppose a diagonal sub-problem solve inside the N^2 loop fails to reach `kOK` or `kLowPrecision`. Then `compute_generic_V_matrix()` throws immediately. It never reaches the `clear()`/`delete` call at the end of the function. Off-diagonal failures do not trigger this. They are caught internally and recorded as the sentinel $V[i][j] = -1.0$.

The loop continues normally (Section 6.2.2). So on a diagonal failure, the Step 1 config clone is neither unregistered from the inner block nor deallocated. This is a known gap in the exception-safety of the class. Some callers catch the exception and continue anyway. They should keep this gap in mind.

6.4 Data Flow Summary

Table 6.2 summarizes the origin of every piece of data consumed by `CSSCScenarioReductionSolver` in the final generic design.

Table 6.2: Data sources used by CSSCScenarioReductionSolver (generic version).

Quantity	**Source**	**Used in**
N (scenarios in pool)	`DiscreteScenarioSet:: get_poolSize()`	Both steps
K (representatives)	`set_nb_reduced(K)` (caller)	Step 2 MILP constraint
Weights $p_i = 1/N$	Uniform, set in `set_Block()`	Step 2 objective; Wasserstein report
Scenario vectors ξ_i	`DiscreteScenarioSet:: get_scenario_value()`	Injected into inner block in Step 1; Euclidean distances in `set_Block()`
First-stage variable pointers	`VarExtractor` callback	Reading y_i^*; fixing/unfixing in Step 1
Inner block (problem structure)	`StochasticBlock` applicator	Solved N^2 times in Step 1
$V[i][j]$ (opportunity costs)	Computed by Step 1	Coefficients of Step 2 MILP
Pairwise Euclidean distances	Computed in `set_Block()` from scenario vectors	Wasserstein report only

The generic design reads all the same information from the `DiscreteScenarioSet`. It also reads through the `VarExtractor` callback. It optionally reads through the `DataInjector` callback too. This keeps the solver's declared class interface independent of any specific block type. As noted in Section 6.2, there is no third hook for block-type quirks. The workaround used for `ECNetworkBlock` is the `f_fix_with_mod` flag, together with the internal `NBModification`-based resynchronisation.

6.5 Summary of the Two Solver Classes

Table 6.3 compares the two solver classes as implemented in the final generic version.

Table 6.3: Summary of the two solver classes in the ScenarioReductionSolver repository.

Property	**ScenarioReductionSolver**	**CSSCScenarioReductionSolver**
Base class	`Solver`	`Solver`
Algorithm	Heuristic (Baseline / Dupacova / BestFit / FirstFit)	CSSC (Step 1 V matrix + Step 2 MILP)
Block type required	`ScenarioReductionBlock` with `DiscreteScenarioSet`	`ScenarioReductionBlock` with `TwoStageStochasticBlock`, `StochasticBlock` applicator, and `DiscreteScenarioSet`
External solver needed	No (pure combinatorial heuristic)	Yes (MILP solver via `BlockSolverConfig`)
Problem-specific code	None (distance computed from scenario vectors)	None (first-stage variables provided via `VarExtractor`; block-type quirks handled via `f_fix_with_mod`, not via problem-specific branches)
CFLB dependency	None	None

Both classes are registered in the SMS++ factory system via `SMSpp_insert_in_factory_cpp_1`, allowing them to be instantiated from a `BlockSolverConfig` file without any direct `new` expressions in the calling code.

Chapter 7

Experimental Framework

This chapter describes the experimental framework shared by the CFL and UC experiments in Chapters 8 and 9. The pipeline, SMS++ components, comparison methods, evaluation metrics, and automation infrastructure are identical across both problem types. Problem-specific details are described in the respective chapters.

7.1 Experimental Pipeline

Every experiment follows the same five-step pipeline, illustrated in Figure 7.1.

Step 1: Load instance and scenarios. The problem instance is loaded from a `netCDF` file via `Block::deserialize()`. The `DiscreteScenarioSet` is deserialised separately. If `-n` is specified, only the first N scenarios are activated via `init_representative_pool(N)`.

Step 2: Solve the full TSS. A `TwoStageStochasticBlock` is assembled from the problem instance and all N scenarios by serialising three components into a temporary `netCDF` file: a `StaticAbstractPath` identifying the first-stage variables, a `StochasticBlock` wrapping a copy of the inner block together with a `DataMapping`, and the `DiscreteScenarioSet`. The assembled block is then deserialised from this file. A solver is attached and `compute()` is called, returning the reference objective Z^* and the solve time.

Step 3: Scenario reduction. A `ScenarioReductionBlock` is created and the `DiscreteScenarioSet` is registered as its scenario generator. For the four heuristic methods, a `ScenarioReductionSolver` is instantiated, the algorithm is selected via `set_algorithm()`, and `compute()` runs the selection procedure. For CSSC, a `CSSCScenarioReductionSolver` is configured with problem-specific callbacks and `compute()` builds the opportunity-cost matrix and solves the MILP partitioning. In both cases, `get_var_solution()` writes the K selected scenario indices, their assignments, and aggregated probability weights into the `ScenarioReductionBlock`.

Step 4: Solve the reduced TSS. A new `DiscreteScenarioSet` is built from the K selected scenario vectors with aggregated weights normalised to sum to 1. A new

`TwoStageStochasticBlock` is assembled from this reduced set and solved to obtain $\tilde{Z}^*$ and the solve time `RedSolve`.

Step 5: Report results. The program prints the full and reduced objectives, the optimality gap, and the two timing measurements.

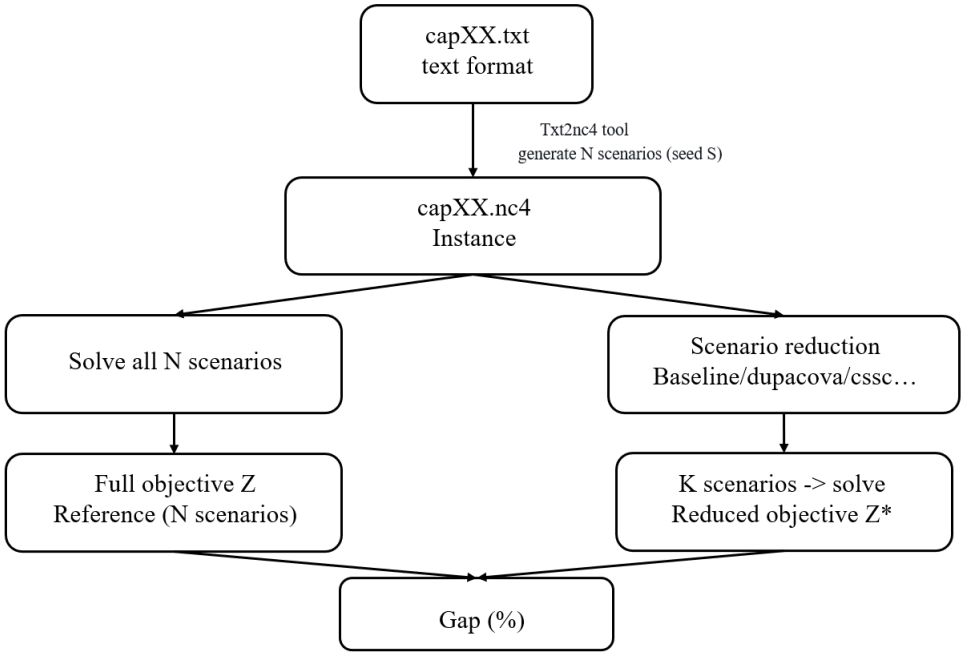

Figure 7.1: Common experimental pipeline.

7.2 SMS++ Components

Four SMS++ components are shared across all experiments regardless of problem type.

`DiscreteScenarioSet` stores the N scenario vectors and their probabilities. It is the sole data source for both solver types. `ScenarioReductionSolver` computes pairwise Euclidean distances directly from the scenario vectors. `CSSCScenarioReductionSolver` reads each scenario vector when building the opportunity-cost matrix during Step 1.

`ScenarioReductionBlock` is the communication object between the `DiscreteScenarioSet` and any attached solver. After `compute()` returns, it holds a `ScenarioReductionBlockSolution` containing the K representative indices, the assignment of every scenario to its representative, and the aggregated probability weights.

`TwoStageStochasticBlock` assembles the extensive form by creating N copies of the inner problem block, injecting one scenario into each copy via a `StochasticBlock` applicator, and linking the first-stage variables via non-anticipativity constraints built from `StaticAbstractPath` objects. It is used twice per experiment: once to produce Z^* and once to produce $\tilde{Z}^*$. `BlockSolverConfig` attaches a MILP solver to any block. The same

configuration file is passed to every executable so that all solves are performed under identical conditions.

7.3 Evaluation Metrics

Five scenario reduction methods are compared. Including Baseline, Dupacova, Bestfit, Firstfit and CSSC. Solution quality is measured by the optimality gap:

$$\text{gap} = \frac{|\tilde{Z}^* - Z^*|}{|Z^*|} \times 100\% \tag{7.1}$$

where Z^* is the optimal value of the full N-scenario problem and $\tilde{Z}^*$ is the optimal value of the reduced K-scenario problem. This metric is reported because it is computable in-sample, directly from the optimal values of the full and reduced problems, without requiring any additional solve. A smaller gap indicates that the reduced problem provides a better approximation of the full stochastic problem.

Computation time is reported in two columns. `AlgoTime` is the wall-clock time spent in the reduction algorithm itself. For CSSC this includes Step 1 (N^2 sub-problem solves) and Step 2 (MILP partitioning). For the heuristics it is only the selection procedure. `RedSolve` is the wall-clock time for solving the reduced K-scenario TSS.

7.4 Solver and Hardware

All executables use `BSPar_CPLEX.txt`, which configures `CPXMILPSolver` with a 3600-second time limit, four threads, and full MIP solving. The same configuration file is passed to every executable, ensuring that Z^* and $\tilde{Z}^*$ are computed under identical conditions and that gap values are directly comparable across methods. CPLEX is chosen because CSSC Step 1 requires solving N^2 independent MIP sub-problems, and CPLEX is substantially faster than open-source alternatives for this workload.

Each problem type has a dedicated automation script that iterates over all combinations of instance, N, K, method, and seed. For each `(instance, N, seed)` combination, scenarios are generated once and shared by all five methods, ensuring that gap values are directly comparable. The full TSS is also solved once per combination and Z^* is reused across all methods at the same (N, seed). Results are appended to a CSV file with the following columns:

```
Instance  N  K  Method  Full_Obj  Reduced_Obj  Gap(%)  AlgoTime(s)  RedSolve(s)
```

All experiments were run on a laptop with an 11th Gen Intel Core i5-1145G7 processor at 2.61 GHz, four physical cores, and 16 GB of RAM. The detail of test results will be described at section 8.5 for the CFL problem and 9.5 for the UC problem.

Chapter 8

Computational Experiments on the CFL Problem

This chapter presents the computational experiments comparing the CSSC algorithm against four heuristic scenario reduction methods on the two-stage stochastic Capacitated Facility Location problem. The experimental framework, pipeline, comparison methods, evaluation metrics, solver configuration, and automation infrastructure are described in Chapter 7. Section 8.1 recalls the problem formulation. Section 8.2 describes the SMS++ components involved. Section 8.3 defines the experimental setup. Section 8.4 presents the CFL-specific test framework. Section 8.5 reports and analyses the results.

8.1 The Capacitated Facility Location Problem

8.1.1 Deterministic Problem

The Capacitated Facility Location (CFL) problem arises in many real-world settings including supply chain management, telecommunications network design, and humanitarian logistics. The goal is to decide which facilities to open from a set of candidate locations in order to serve a group of customers at minimum total cost. The total cost consists of fixed opening costs and variable transportation costs.

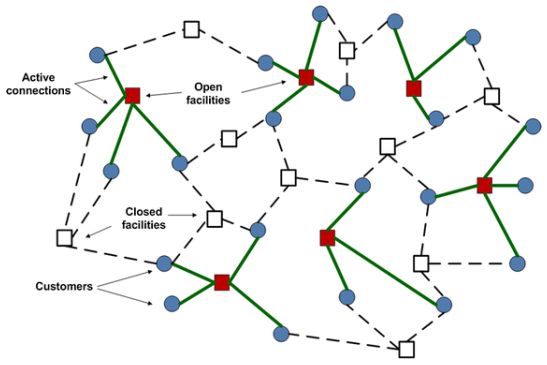

Figure 8.1: Capacitated Facility Location problem example.

In practice, these decisions involve long-term investments that are difficult to reverse. At the same time, key parameters such as customer demand are often uncertain due to market volatility, seasonal changes, or unexpected disruptions. The deterministic version of the problem assumes all parameters are known with certainty. We present it first as the basis for the stochastic extension in the next section.

Note that in the CFL formulation, we follow the standard notation of the facility location literature, where y_i denotes the facility opening decisions and x_{ij} denotes the allocation variables. We define I as the set of candidate facility locations and J as the set of customers. For each facility $i \in I$, there is a fixed opening cost F_i and a maximum supply capacity K_i. The transportation cost from facility i to customer j is C_{ij} and the demand of customer j is d_j. The binary variable $y_i \in \{0, 1\}$ indicates whether facility i is opened. The continuous variable $x_{ij} \geq 0$ represents the quantity shipped from facility i to customer j. The deterministic model is:

$$\min \sum_{i \in I} F_i y_i + \sum_{i \in I} \sum_{j \in J} C_{ij} x_{ij} \tag{8.1}$$

$$\begin{aligned} \text{s.t.} \quad & \sum_{i \in I} x_{ij} = d_j, && \forall j \in J \\ & \sum_{j \in J} x_{ij} \leq K_i y_i, && \forall i \in I \\ & x_{ij} \geq 0,\ y_i \in \{0, 1\}, && \forall i \in I,\ \forall j \in J \end{aligned} \tag{8.2}$$

The demand satisfaction constraint ensures the total amount shipped to each customer exactly meets their demand. The capacity constraint ensures the total outflow from a facility does not exceed its capacity and also enforces that shipments can only originate from an open facility.

8.1.2 Two-Stage Stochastic Model

In practice, facility opening decisions involve long-term investments that cannot be easily reversed, yet customer demand is frequently uncertain. We adopt a two-stage stochastic formulation with a set of discrete scenarios Ω, where each scenario $\omega \in \Omega$ occurs with probability p_ω and $\sum_{\omega \in \Omega} p_\omega = 1$. Customer demand is scenario-dependent, denoted $d_{j\omega}$. The shipment variable is adjusted to $x_{ij\omega}$, while the facility opening variable y_i remains the same across all scenarios. The two-stage stochastic CFL is:

$$\min \sum_{i \in I} F_i y_i + \sum_{\omega \in \Omega} p_\omega \left(\sum_{i \in I} \sum_{j \in J} C_{ij}\, x_{ij\omega} \right) \tag{8.3}$$

$$\begin{aligned} \text{s.t.} \quad & \sum_{i \in I} x_{ij\omega} = d_{j\omega}, && \forall j \in J,\ \forall \omega \in \Omega \\ & \sum_{j \in J} x_{ij\omega} \leq K_i y_i, && \forall i \in I,\ \forall \omega \in \Omega \\ & y_i \in \{0, 1\},\ x_{ij\omega} \geq 0, && \forall i, j, \omega \end{aligned} \tag{8.4}$$

When $|\Omega|$ is large, solving this extensive form directly becomes computationally expensive, which motivates the use of scenario reduction as described in Chapter 3.

8.2 SMS++ Components for the CFL Experiments

`CapacitatedFacilityLocationBlock` is the inner block encoding the CFL model. It stores all instance data and exposes the binary facility opening variables y_i as the first-stage variables, identified by `AbstractPath` objects inside `TwoStageStochasticBlock`. `DiscreteScenarioSet` stores the N demand scenarios and their probabilities. It is the sole data source for both solvers: `ScenarioReductionSolver` computes pairwise Euclidean distances directly from the scenario vectors, and `CSSCScenarioReductionSolver` injects each scenario into the inner CFL block during Step 1 via the `DataInjector` callback.

`ScenarioReductionBlock` acts as the communication object between the `DiscreteScenarioSet` and any attached solver. After `compute()` returns, it holds the `ScenarioReductionBlockSolution` containing the K representative indices, assignments, and aggregated weights. `TwoStageStochasticBlock` assembles the extensive form by creating N copies of the CFL block, injecting one scenario into each copy, and linking the facility opening variables via non-anticipativity constraints. It is used twice: once to produce the reference objective Z^* with all N scenarios, and once to produce the reduced objective $\tilde{Z}^*$ with the K selected representatives.

8.3 Experimental Setup

8.3.1 Instances

Experiments are conducted on two standard OR-Library instances with varying problem scale.

Table 8.1: CFL instances used in the experiments.

Instance	**Facilities**	**Customers**	**Scale**
cap71	16	50	Medium
cap121	50	50	Large

The two instances differ in the number of candidate facility locations and in their capacity structure. This leads to distinct behaviour across methods: larger instances have more complex first-stage decisions and larger recourse problems, which tends to make the differences between distribution-driven and cost-driven reduction methods more pronounced.

8.3.2 Scenario Generation

Demand scenarios are generated by `CFLScenarioGenerator` using a clustered perturbation strategy. The first scenario is the unmodified base demand vector. The remaining $N - 1$ scenarios are distributed equally across five demand clusters, each defined by a multiplicative perturbation applied independently to every customer demand d_j:

Table 8.2: Demand clusters used in scenario generation.

Cluster	**Multiplier**	**Description**
Normal	$U(0.9,\ 1.1)$	Scenarios close to base demand.
High	$U(1.3,\ 1.5)$	Uniformly elevated demand.
Low	$U(0.5,\ 0.7)$	Uniformly reduced demand.
Mixed	$U(1.2,\ 1.4)$ / $U(0.6,\ 0.8)$	High for first half of customers, low for second.
Very low	$U(0.4,\ 0.6)$	Demand reduced to approximately half.

All scenarios are assigned equal probability $1/N$. Each generated scenario is validated by solving the CFL instance with a 10-second time limit, any infeasible scenario is regenerated with a milder perturbation. The clustered structure produces a scenario set with multiple distinct demand regimes, which is more challenging for scenario reduction than a unimodal distribution and better reflects the kind of variability that problem-driven methods such as CSSC are designed to exploit.

8.3.3 Parameter Ranges

Table 8.3: Experimental parameter values.

Parameter	**Values**	**Description**
N	20, 50, 100	Number of scenarios in the full set. Controls problem size and the difficulty of the reduction task.
K	5, 10	Number of representative scenarios after reduction.
Seeds	1–10	10 independent seeds per combination to assess result stability across different scenario sets.

Table 8.3 summarizes the parameter grid used in the experiments. For each combination of instance, N, K, and seed, all five methods share the same scenario set and the same full TSS solve, ensuring that gap values are directly comparable. The solver configuration follows section 7.4.

8.4 Test Framework

8.4.1 Overview

The test program is implemented in `CFLScenarioReductionTest.cpp` as a self-contained `main()` function with two helper functions. `build_tssb()` constructs a `TwoStageStochastic Block` from a CFL instance and a `DiscreteScenarioSet` by serialising the required components into a temporary `netCDF` file and deserialising the assembled block from it. `solve_tssb()` generates the abstract representation, attaches the solver, calls `compute()`, and returns the objective value and elapsed time.

The program accepts the following command-line arguments:

```
-i <file>   CFL instance netCDF file (required)
-f <file>   Scenario netCDF file (required)
-r <int>    Number of representatives K (default: 5)
-n <int>    Limit total scenarios to N (default: all)
-c <file>   Solver config file (default: BSPar_CPLEX.txt)
-m <method> dupacova|bestfit|firstfit|baseline|cssc
-v <int>    Verbosity (default: 0)
```

The same executable handles all five methods via the `-m` flag.

8.4.2 Pipeline

The program follows the five-step pipeline described in Chapter 7. In Step 1, the instance is cast to `CapacitatedFacilityLocationBlock` and the `UnSplittable` flag is set to `true` to enforce single-sourcing. The number of facilities `nf` and customers `nc` are read from the block. In Step 2, `build_tssb()` identifies the `nf` facility opening variables y_i as the non-anticipativity variables via a `StaticAbstractPath`, and the `DataMapping` maps scenario vector positions $[0, \texttt{nc})$ to customer demands. In Step 3, the two solver paths diverge. For the heuristic methods, `ScenarioReductionSolver` reads pairwise Euclidean distances directly from the `DiscreteScenarioSet` without requiring any additional block. For CSSC, a `TwoStageStochasticBlock` is first built from the CFL instance and all N scenarios.

The `StochasticBlock` applicator is retrieved and registered with the `ScenarioReduction Block` via `set_scenario_applicator()`. A `CSSCScenarioReduction Solver` is then created and configured:

CSSC setup in CFLScenarioReductionTest.cpp

```
auto cssc = std::make_unique< CSSCScenarioReductionSolver >();
cssc→set_milp_config(
 static_cast< BlockSolverConfig * >( bsc→clone() ) );
cssc→set_nb_reduced( K );
cssc→set_Block( srb.get() );

// VarExtractor: returns the nf facility opening variables y[f]
```

```
cssc->set_var_extractor( []( Block * inner ) {
 auto * cflb =
  dynamic_cast< CapacitatedFacilityLocationBlock * >( inner );
 if( ! cflb ) return std::vector< ColVariable * >{};
 const int nf = static_cast< int >( cflb->get_NFacilities() );
 std::vector< ColVariable * > vars;
 for( int f = 0 ; f < nf ; ++f )
  vars.push_back( cflb->get_y( f ) );
 return vars;
} );

// DataInjector: notifies the solver after set_data()
// because the DataMapping uses eNoMod and does not
// send solver notifications automatically
cssc->set_data_injector( []( Block * inner ,
                             const std::vector< double > & data ) {
 auto * cflb =
  dynamic_cast< CapacitatedFacilityLocationBlock * >( inner );
 if( ! cflb ) return;
 using CIdx = CapacitatedFacilityLocationBlock::Index;
 cflb->chg_customer_demands( data.begin() ,
  Block::Range( CIdx(0) , CIdx( data.size() ) ) ) ,
  eNoMod , eModBlck );
} );

cssc->compute();
cssc->get_var_solution();
```

The `VarExtractor` callback is the only CFL-specific piece of code in the CSSC setup. It tells the solver which variables are the first-stage decisions, so that their optimal values y_i^* can be read after the diagonal MIP solve and fixed during the $N-1$ off-diagonal LP solves in Step 1 of the CSSC algorithm. The `DataInjector` handles a subtlety specific to the CFL block: after `StochasticBlock::set_data()` updates the physical demand values, it does not automatically notify the attached MILP solver because the `DataMapping` was configured with `eNoMod` flags. The injector calls `chg_customer_demands(eModBlck)` explicitly to propagate the update before the next `compute()` call.

After `compute()` and `get_var_solution()` return, the `ScenarioReductionBlockSolution` inside the `ScenarioReductionBlock` contains the K selected scenario indices, the assignment of every scenario to its representative, and the aggregated probability weights. The time spent in the reduction algorithm is recorded at this point as `RedAlgo`. Steps 4 and 5 proceed as described in Chapter 7.

8.5 Results and Analysis

The script `run cfl test.sh` automates the full comparison pipeline. We organize the analysis by scenario pool size N , since it is the parameter that most directly governs

both the computational cost of CSSC and the gap achieved by all methods. The results are mean values over 10 seeds.

8.5.1 Small Scale: $N = 20$

Table 8.4 reports the mean in-sample gap at $N = 20$. The best results of each row are in bold.

Table 8.4: Mean in-sample gap (%) at $N = 20$.

Instance	K	Baseline	Dupacova	BestFit	FirstFit	CSSC
cap71	5	0.241	0.052	0.056	0.044	**0.032**
cap71	10	0.241	0.051	0.044	0.053	**0.000**
cap121	5	1.028	0.128	0.115	0.132	**0.070**
cap121	10	1.028	0.128	0.166	0.169	**0.000**

At $N = 20$, CSSC already outperforms all four heuristics at both values of K. At $K = 10$, CSSC recovers the full-problem objective almost exactly on both instances (gap approximately 0%), while the heuristics remain in the 0.04–0.24% range on `cap71` and 0.13–0.17% on `cap121`. Unlike the demand-clustered instances analyzed in Section 8.3, the four heuristics no longer produce identical gaps here: Dupacova, BestFit, and FirstFit diverge slightly from one another, while Baseline remains consistently the worst performer, confirming that selecting by probability weight alone provides no useful signal when probabilities are uniform.

Table 8.5: Mean `AlgoTime` (reduction step only) for $N = 20$.

Instance	N	K	Heuristics (ms)	CSSC (s)	Overhead
cap71	20	5	0.006 – 0.006	0.844	∼140×
cap71	20	10	0.010 – 0.012	0.831	∼69×
cap121	20	5	0.038 – 0.045	2.506	∼56×
cap121	20	10	0.101 – 0.116	2.671	∼23×

On `cap121`, where facility costs and capacities introduce a harder second-stage structure, the gap advantage of CSSC over the best heuristic is modest at $N = 20$, around 1.6× at $K = 5$. The harder instance evidently makes good representative scenarios harder to find for every method, including CSSC, narrowing the relative gap between approaches at this small pool size.

8.5.2 Medium Scale: $N = 50$

Table 8.6 reports the mean in-sample gap at $N = 50$. The best results of each row are in bold.

Table 8.6: Mean in-sample gap (%) at $N = 50$.

Instance	K	Baseline	Dupacova	BestFit	FirstFit	CSSC
cap71	5	0.203	0.077	**0.053**	0.077	0.056
cap71	10	0.203	0.045	0.039	0.035	**0.023**
cap121	5	0.968	0.144	0.156	0.144	**0.139**
cap121	10	0.968	0.102	0.086	0.073	**0.052**

At $N = 50$, CSSC's advantage over the heuristics narrows on `cap71`. At $K = 5$, BestFit (0.053%) is marginally better than CSSC (0.056%), meaning the best heuristic and CSSC are essentially tied at this setting. On `cap121` at $K = 5$, CSSC (0.139%) remains slightly ahead of the best heuristic (0.144%), but the margin is small. At $K = 10$, CSSC recovers its lead on both instances, achieving a gap roughly 1.5× lower than the best heuristic on `cap71` (0.023% vs. 0.035%) and 1.4× lower on `cap121` (0.052% vs. 0.073%).

Table 8.7: Mean `AlgoTime` (reduction step only) for $N = 50$.

Instance	N	K	Heuristics (ms)	CSSC (s)	Overhead
cap71	50	5	0.009 – 0.010	4.352	∼440×
cap71	50	10	0.014 – 0.020	4.479	∼224×
cap121	50	5	0.037 – 0.040	12.365	∼309×
cap121	50	10	0.102 – 0.117	12.548	∼107×

The computational cost of CSSC grows substantially between $N = 20$ and $N = 50$: `AlgoTime` increases by a factor of approximately 5.2–5.4× on `cap71` and 4.7–4.9× on `cap121`, both close to but somewhat above the $O(N^2)$ prediction of 2.5× for this specific step (N going from 20 to 50 is a 2.5× increase, so N^2 predicts 6.25×; the observed growth is in fact slightly below this quadratic bound). The heuristics, by contrast, remain effectively free: their `AlgoTime` stays under 0.12 ms regardless of N or instance.

8.5.3 Large Scale: $N = 100$

Table 8.8 reports the mean in-sample gap at $N = 100$. The best results of each row are in bold.

Table 8.8: Mean in-sample gap (%) at $N = 100$.

Instance	K	Baseline	Dupacova	BestFit	FirstFit	CSSC
cap71	5	0.188	**0.067**	**0.067**	**0.067**	0.083
cap71	10	0.209	0.075	0.063	0.063	**0.061**
cap121	5	0.599	0.134	0.134	0.134	**0.113**
cap121	10	0.599	0.107	0.118	0.112	**0.060**

At $N = 100$, the picture changes qualitatively on `cap71`. At $K = 5$, CSSC (0.083%) is for the first time worse than the distribution-driven heuristics (0.067% for Dupacova, BestFit, and FirstFit), and at $K = 10$ the gap of all methods has converged to a narrow 0.061–0.075% band where CSSC holds only a marginal lead. This is consistent with the most aggressive reduction ratio ($K = 5$ out of $N = 100$, i.e. 5%) being the hardest case for the MILP partitioning step of CSSC to resolve well, as already observed in the per-instance analysis on the larger OR-Library instances. On `cap121`, CSSC retains a clear advantage at both $K = 5$ (0.113% vs. 0.134%) and especially $K = 10$ (0.060% vs. 0.107–0.118%, roughly 1.8–2.0× better), suggesting that on the harder, more capacity-constrained instance the cost-space information exploited by CSSC remains valuable even as N grows, while on the easier `cap71` instance the distribution-driven heuristics catch up.

Table 8.9: Mean `AlgoTime` (reduction step only) for $N = 100$.

Instance	N	K	Heuristics (ms)	CSSC (s)	Overhead
cap71	100	5	0.027 – 0.032	23.211	∼725×
cap71	100	10	0.023 – 0.038	30.465	∼1,009×
cap121	100	5	0.053 – 0.055	97.304	∼1,769×
cap121	100	10	0.124 – 0.131	123.936	∼945×

The computational cost of CSSC continues to grow steeply: `AlgoTime` reaches 23–30 s on `cap71` and 97–124 s on `cap121` at $N = 100$. The growth from $N = 50$ to $N = 100$ is 5.3–6.8× on `cap71` and 7.9–9.9× on `cap121`, both exceeding the $O(N^2)$ prediction of 4× for a doubling of N. This super-linear growth indicates that the individual single-scenario CFL sub-problems solved in CSSC Step 1 become harder, not just more numerous, as N increases. The overhead ratio of CSSC `AlgoTime` relative to the (essentially negligible) heuristic `AlgoTime` is correspondingly large, reaching roughly 800–1,000× on `cap71` and 1,800× on `cap121` at $N = 100$.

8.5.4 Computational Time

The gap between CSSC and the heuristics is not a matter of constant factors or implementation inefficiency, it follows directly from what each method computes. The four

heuristics are purely distribution-driven: they operate on a precomputed $N \times N$ matrix of pairwise Euclidean distances between scenario vectors and apply a combinatorial selection rule (highest weight, forward selection, or local search). No optimization sub-problem is solved at any point. The dominant cost is $O(N^2)$ floating-point distance evaluations, each of which is essentially free on instances of this size, which is why `AlgoTime` for the heuristics remains in the microsecond-to-millisecond range even at $N = 100$.

CSSC, by contrast, is problem-driven. Its Step 1 builds the opportunity-cost matrix V by solving N full mixed-integer CFL sub-problems (the diagonal V_{ii}) and $N(N-1)$ linear sub-problems with the first-stage facility variables fixed (the off-diagonal V_{ij}), for a total of N^2 calls to a MILP/LP solver. Each of these calls, however small the instance, involves the overhead of a solver invocation: building or updating the constraint matrix, performing branch-and-bound for the diagonal MIP solves, and re-optimising the LP relaxation for the fixed-design solves.

On top of the N^2 sub-problem count, the growth observed in `AlgoTime` consistently exceeds the $O(N^2)$ prediction. Going from $N = 50$ to $N = 100$, a $4\times$ increase in N^2, the observed increase in CSSC's `AlgoTime` is 5.3–6.8$\times$ on `cap71` and 7.9–9.9$\times$ on `cap121`. This excess indicates that the individual sub-problems are not just more numerous as N grows, but individually harder to solve: a larger scenario pool spans a wider range of demand realisations, which increases the diversity of facility configurations that must be explored by the branch-and-bound solver and weakens the warm-start benefit carried over from one sub-problem to the next. Step 2, the MILP partitioning problem with $O(N^2)$ binary assignment variables, compounds this further at larger N, although the dominant cost throughout remains Step 1, consistent with the K-insensitivity of `AlgoTime` observed when K is varied at fixed N: doubling K from 5 to 10 changes `AlgoTime` by at most a few percent, while increasing N from 20 to 50 (a 2.5$\times$ increase) raises it by a factor of five or more.

The cost structure observed here has a direct practical consequence for when CSSC is worth using. If the reduction is computed once and the resulting representative set is reused across many downstream solves (for example, if the same reduced scenario set is used to evaluate several candidate facility configurations, or embedded inside an outer optimization loop), the one-time cost of CSSC's `AlgoTime` is amortised and its lower gap makes it the preferable choice whenever it actually achieves a meaningfully lower gap than the heuristics. If, instead, the reduction is needed only once and N is large, the heuristics offer a gap that is often competitive, as seen at $N = 100$, $K = 5$ on `cap71`, while completing in milliseconds rather than tens of seconds. The crossover point depends on both the instance difficulty, since the harder `cap121` instance preserves CSSC's gap advantage at larger N than the easier `cap71`, and on how many times the reduced scenario set will be reused after it is constructed.

Across both instances and all values of N and K, the reduced TSS solve time (`RedTime`) is comparable across all five methods at the same (N, K) setting, since all methods produce a reduced problem of identical size K. The full details are represented in the table 8.10 and 8.11. A notable exception is `cap121` at $K = 10$, where CSSC produces a faster reduced problem than the heuristics (e.g. 0.797 s versus 1.20–1.28 s at $N = 50$, and 1.158 s versus 1.51–1.76 s at $N = 100$, a difference of roughly 35–40%). This suggests that the

representative set selected by CSSC is occasionally somewhat easier for the second-stage solver, in addition to producing a lower gap.

Table 8.10: Reduced Problem Solve Time (`RedTime`) in seconds for `cap71`

N	K	baseline	dupacova	bestfit	firstfit	cssc
20	5	0.054	0.055	0.051	0.051	**0.049**
	10	0.106	0.098	**0.096**	0.105	0.097
50	5	0.059	0.056	0.054	0.058	**0.051**
	10	0.115	0.114	0.116	0.127	**0.098**
100	5	0.069	0.072	0.072	0.074	**0.054**
	10	0.123	0.120	0.117	0.117	**0.100**

Table 8.11: Reduced Problem Solve Time (`RedTime`) in seconds for `cap121`

N	K	baseline	dupacova	bestfit	firstfit	cssc
20	5	0.610	0.641	0.691	0.696	**0.528**
	10	1.205	1.366	1.354	1.302	**1.106**
50	5	0.648	0.657	0.605	0.634	**0.511**
	10	1.203	1.237	1.280	1.253	**0.797**
100	5	0.949	0.986	0.978	0.953	**0.916**
	10	1.597	1.762	1.513	1.594	**1.158**

It is also worth noting that on the CFL instances tested here, `RedTime` remains small relative to CSSC's `AlgoTime` across all settings. The total solution time is therefore dominated by the reduction step itself, not by the reduced problem it produces. On larger or harder instances where `RedTime` grows significantly, this balance may shift, making CSSC more competitive in terms of total running time.

To further explore this trade-off, we extend the experiments to $K \in \{20, 30\}$ at $N = 100$. Table 8.12 and 8.13 report the results.

Table 8.12: Total time and in-sample gap (%) at $N = 100$, $K \in \{20, 30\}$ for `cap71`.

K	Method	Gap (%)	AlgoTime (s)	RedTime (s)	Total (s)
20	Baseline	0.440	0.016	0.215	0.231
	Dupacova	0.018	0.018	0.217	0.236
	BestFit	0.109	0.040	0.211	0.250
	FirstFit	0.051	0.039	0.199	0.239
	CSSC	0.034	25.789	0.203	25.993
30	Baseline	0.019	0.019	0.356	0.375
	Dupacova	0.018	0.024	0.376	0.401
	BestFit	0.037	0.089	0.406	0.495
	FirstFit	0.041	0.102	0.426	0.528
	CSSC	0.000	20.070	0.340	20.410

Table 8.13: Total time and in-sample gap (%) at $N = 100$, $K \in \{20, 30\}$ for `cap121`.

K	Method	Gap (%)	AlgoTime (s)	RedTime (s)	Total (s)
20	Baseline	1.164	0.022	2.549	2.570
	Dupacova	0.128	0.024	2.595	2.619
	BestFit	0.149	0.062	3.729	3.791
	FirstFit	0.187	0.084	4.538	4.622
	CSSC	0.002	166.111	1.676	167.787
30	Baseline	0.233	0.021	3.772	3.793
	Dupacova	0.122	0.026	4.500	4.526
	BestFit	0.153	0.113	4.620	4.733
	FirstFit	0.107	0.080	3.581	3.662
	CSSC	0.002	165.919	3.112	169.031

As K increases, CSSC's `AlgoTime` stays flat while `RedTime` grows for all methods. On `cap121`, the best heuristic total time grows from 1.5 s at $K = 10$ to 3.7 s at $K = 30$, while CSSC total grows from 125 s to 169 s. The time overhead of CSSC remains large on these small instances, but this trend suggests that on larger and more complex problems, `RedTime` could eventually dominate the total time, narrowing the difference between CSSC and the heuristics. In terms of solution quality, CSSC's optimality gap continues to improve with K, while the heuristics show more modest improvements.

8.5.5 Conclusion

Three observations emerge from varying N. First, CSSC's advantage over the heuristics is largest at $N = 20$, where the gap reduction is most dramatic: at $K = 10$, CSSC recovers almost exactly the full-problem objective on both instances. With a small scenario pool, the reduction problem is easier to solve well, giving CSSC's cost-space objective more room to find a near-optimal partition.

Second, this advantage shrinks steadily as N grows from 20 to 100 on the easier instance `cap71`. At the most aggressive reduction ratio ($K = 5$, $N = 100$), the heuristics become competitive with CSSC, and even slightly better. On the harder instance `cap121`, however, CSSC keeps a clear advantage at $K = 10$ even at $N = 100$, suggesting that CSSC's benefit is more persistent when second-stage costs are more sensitive to the choice of representative scenarios.

Third, CSSC's computational cost grows faster than N^2, meaning the sub-problems get harder to solve as the scenario pool grows. For easier instances and large N, this extra cost may not always pay off. CSSC is most worthwhile when N is moderate or when the instance has a harder second-stage structure.

Regarding the trade-off between K and total solution time, increasing K improves solution quality for all methods but also increases `RedTime`. Since CSSC's `AlgoTime` stays flat as K grows, its total time grows more slowly than the heuristics' total time at larger K. This effect is more visible on `cap121`, where `RedTime` is non-negligible, and suggests that CSSC becomes more competitive in total running time on harder instances with larger recourse problems.

Finally, beyond solution quality, CSSC identifies the most important scenarios from a cost perspective. This can provide valuable insight to decision-makers, helping them understand which combinations of conditions are most critical to consider.

Chapter 9

Computational Experiments on the UC Problem

This chapter presents the computational experiments comparing the CSSC algorithm against four heuristic methods on the two-stage stochastic Unit Commitment (UC) problem, specifically on Energy Community (EC) instances. The experimental framework, pipeline, comparison methods, evaluation metrics, solver configuration, and automation infrastructure are described in Chapter 7. Section 9.1 recalls the EC formulation. Section 9.2 describes the SMS++ components involved. Section 9.3 presents the test framework in detail. Section 9.4 defines the experimental setup. Section 9.5 reports and analyses the results.

9.1 The Two-Stage Stochastic Energy Community Problem

Among the Unit Commitment subtypes supported by SMS++, this work focuses on the Energy Community (EC) case, modelled with `ECNetworkBlock` as the network component inside `UCBlock`. In an Energy Community, a set of users can buy energy from the public market, sell surplus energy back, and share energy within the community through a microgrid. A peak tariff applies based on the maximum net power exchanged at the aggregation point over the planning horizon.

The first-stage decisions are those that must be fixed before demand is known. In the EC context, these are primarily the commitment variables $x_{it} \in \{0, 1\}$ of each thermal generating unit i at each time period t, indicating whether the unit is on or off. These must be decided in advance and are linked across all scenario copies via non-anticipativity constraints.

The second-stage decisions are optimized independently for each scenario ξ_k once the demand is revealed. They include at least two families of variables. The first family consists of the production variables of each generating unit: the power output $p_{it}^k \geq 0$ of each unit, the dispatch of fuel-fired generators $P_{j,t}^{g,k}$, the output of renewable sources $P_{j,t}^{R,k}$, and the charging/discharging of battery converters. The second family consists of the EC

network variables inside `ECNetworkBlock`: the power injection $P^{+,k}_{n,t}$, power absorption $P^{-,k}_{n,t}$, shared power $P^{M,k}_{n,t}$, and peak power $P^{\max,k}_{n}$ at each node.

Importantly, the second-stage problem is in general a mixed-integer program, not a pure linear program. Some second-stage variables are integer: specifically, the on/off scheduling variables $z^{g,k}_{j,t} \in \{0,1\}$ of fuel-fired generators within the community determine the operating state of each generator in the real-time scenario. This is a fundamental structural difference from the Capacitated Facility Location problem, where the recourse is always a pure LP once the first-stage variables are fixed.

The resulting two-stage stochastic problem minimises the sum of first-stage commitment costs and the expected second-stage operating cost:

$$\min \; f(x) + \sum_{k=1}^{N} p_k \, Q(x, \, \xi_k) \tag{9.1}$$

where $f(x)$ contains the start-up and shut-down costs of all thermal units, p_k is the probability of scenario k, and $Q(x, \xi_k)$ is the optimal recourse cost under scenario k with commitment x fixed. For each scenario k, the recourse objective aggregates the EC network costs and the production costs of the units:

$$Q(x, \, \xi_k) = \min \sum_{j,t} \Big[C^{g,k}_{j,t} + \pi^{-}_{t} P^{-,k}_{n,t} - \pi^{+}_{t} P^{+,k}_{n,t} - \pi^{r}_{t} P^{M,k}_{n,t} \Big] + \sum_{n} \pi^{\max} P^{\max,k}_{n} \tag{9.2}$$

subject to the per-node power balance, peak power constraints, generator dispatch constraints, and battery energy balance for each scenario k, where $C^{g,k}_{j,t}$ includes the generation and fuel costs of fuel-fired generators, and the `ActiveDemand` matrix takes the value ξ_k. The non-anticipativity constraint enforces that the first-stage commitment decisions x_{it} are identical across all N scenario copies.

The uncertain parameter is the customer demand encoded in the `ActiveDemand` matrix of `ECNetworkBlock`, of shape [`NumberIntervals` $\times$ `NumberNodes`]. When $|\Omega|$ is large, solving this extensive form directly becomes computationally expensive, motivating scenario reduction as discussed in Chapter 3.

9.2 SMS++ Components for the EC Experiments

`UCBlock` is the inner block encoding the Unit Commitment model. It holds a collection of generating units, each a concrete subclass of `UnitBlock`, as well as a network component for the EC instances. For the EC experiments in this chapter, the generating units are `ThermalUnitBlock` instances, whose commitment variables $x_{it} \in \{0,1\}$ serve as the first-stage decisions, identified by `StaticAbstractPath` objects inside `TwoStageStochasticBlock`.

`ECNetworkBlock` holds the `ActiveDemand` matrix, the uncertain parameter that changes across scenarios. It is not a standalone block but a network component inside `UCBlock`, the full second-stage problem includes both the EC network variables of `ECNetworkBlock` and the production variables of the generating units. Unlike `CapacitatedFacilityLocation`

`Block`, its incremental data-update path has a known limitation for instances where the number of nodes differs from the number of intervals, which directly affects how scenarios are injected during CSSC Step 1 (Section 9.3.4).

`DiscreteScenarioSet` stores the N demand scenarios and their probabilities, exactly as in the CFL case. It is the sole data source for both solvers. `ScenarioReductionBlock` acts as the communication object between the `DiscreteScenarioSet` and any attached solver, identical in role to its use in the CFL experiments.

`TwoStageStochasticBlock` assembles the extensive form by creating N copies of the `UCBlock`, injecting one scenario into each copy, and linking the commitment variables x_{it} via non-anticipativity constraints. It is used twice per experiment: once to produce the reference objective Z^* with all N scenarios, and once to produce the reduced objective $\tilde{Z}^*$ with the K selected representatives.

9.3 Test Framework

9.3.1 Overview

The UC test program is implemented as a set of free functions. They live directly in `SMSpp_di_unipi_it`, the main SMS++ namespace. The functions sit alongside the rest of the SMS++ code instead. The four key functions are: `uc_load_problem_instance()`, `uc_create_srb()`, `uc_build_tssb_for_current_pool()`, and `uc_run_cssc()`.

Four of the hooks carry problem-specific logic:

- `load_problem_instance()`: loads the base `UCBlock` instance and wraps it in a `StochasticBlock`.
- `create_srb()`: builds a fully configured `ScenarioReductionBlock`, including the `TwoStageStochasticBlock` and applicator when the method is CSSC.
- `build_tssb_for_current_pool()`: builds a `TwoStageStochasticBlock` from the current scenario pool.
- `run_cssc()`: configures and runs `CSSCScenarioReductionSolver` with UC-specific callbacks.

Everything else is implemented once, inside `run_scenario_reduction_test()`. That includes caching of intermediate results, warm-start logic, and result printing. At that level, the CFL and UC test programs are therefore identical.

9.3.2 Loading the Instance

`uc_load_problem_instance()` loads a `UCBlock` from a `netCDF` file. This test builds the `TwoStageStochasticBlock` itself, from a plain `UCBlock` and a scenario set. So it needs a bare `UCBlock` file as input. Sometimes the supplied path carries a `TSSB_` prefix instead: this is added when a temporary extensive-form file is written, as described below. When that happens, the prefix is stripped before deserialisation, so the underlying bare instance file is loaded instead. The dataset used in these experiments, `EC_CO_Test_TUB.nc4`, follows this convention directly. After loading, the function records the time horizon, the number

of nodes, and the indices of any `IntermittentUnitBlock` children. These children are used for renewable uncertainty in other UC subtypes, they are not exercised in the EC experiments here. A `StochasticBlock` wrapper is then constructed around the loaded `UCBlock`.

`UCBlock` can carry both demand and renewable uncertainty. So the test framework infers which type is present. It does this directly from the dimension of the loaded scenario vectors, via `infer_uncertainty_type()`. This inference does not happen at load time itself: the scenario pool has not yet been attached when `uc_load_problem_instance()` runs. It is instead performed later, each time a `TwoStageStochasticBlock` or `ScenarioReductionBlock` is (re)built from the current pool. Three expected dimensions are compared against the actual scenario size, given the time horizon T, the number of nodes `nd`, and the number of intermittent units `ni`: `nd` $\times T$ for demand-only, `ni` $\times T$ for renewable-only, and their sum for both. The EC experiments in this chapter use demand-only scenarios.

9.3.3 Building the TwoStageStochasticBlock

A shared internal helper, `build_tssb()`, assembles a `TwoStageStochasticBlock` from the `UCBlock` and a given scenario pool. It follows a serialise-then-deserialise pattern. It is used by `uc_build_tssb_for_current_pool()`, to solve the full and reduced TSS. For the CSSC method, it is also used by `uc_create_srb()`, to build the applicator TSSB handed to the `ScenarioReductionBlock`. Three components are written to a temporary `netCDF` file: a `StaticAbstractPath` identifying the commitment variables x_{it} of every `ThermalUnitBlock` as the non-anticipativity variables, a `StochasticBlock` wrapping a serialised copy of the `UCBlock` together with the appropriate `DataMapping`, and the `DiscreteScenarioSet`.

The `DataMapping` for demand uncertainty targets `UCBlock::set_active_power_demand`, mapping scenario vector positions $[0, \texttt{nd}\times T)$ to the corresponding entries of the `ActiveDemand` matrix.

9.3.4 Running CSSC on UC: VarExtractor and Reload Mode

`uc_run_cssc()` configures `CSSCScenarioReductionSolver` with a single UC-specific callback, the `VarExtractor`. It also sets one configuration flag. The `VarExtractor` returns the commitment variables x_{it} of every `ThermalUnitBlock`, across all T time periods:

UC VarExtractor: commitment variables x_{it} of every ThermalUnitBlock

```
solver->set_var_extractor( [T]( Block * inner ) ->std::vector< ColVariable
    * > {
 auto * uc = dynamic_cast< UCBlock * >( inner );
 if( ! uc ) return {};
 std::vector< ColVariable * > vars;
 const Index num_units = uc->get_number_units();
 for( Index u = 0 ; u < num_units ; ++u ) {
  auto * tub = dynamic_cast< ThermalUnitBlock * >( uc->get_unit_block(u) );
  if( tub ) {
```

```
    ColVariable * cv = tub→get_commitment( 0 );
    if( cv )
     for( Index t = 0 ; t < T ; ++t )
      vars.push_back( cv + t );
    }
   }
  return vars;
} );
```

No `DataInjector` is registered. Instead, the solver is switched into reload mode. A single boolean flag does this: `set_fix_with_modification(false)`:

Reload mode: UCBlock cannot notify its solver of incremental changes

```
// UCBlock cannot tell its Solver about every incremental data
// change, so CSSC must use reload mode: after each set_data(), the whole
// model is reloaded from scratch
solver→set_fix_with_modification( false );
```

This addresses the same underlying limitation introduced in Section 9.2. `ECNetworkBlock`'s incremental data-update path cannot be trusted. It fails to keep the abstract model and the attached solver in sync, when the number of nodes differs from the number of intervals. Inside `CSSCScenarioReductionSolver`, this flag selects which branch of the internal `inject_scenario()` lambda is taken (see the class implementation in Chapter 6, Section 6.2). When it is `true`, the CFL default, `set_data()` is called with `eNoBlck` on both ends. A caller-supplied `DataInjector` would then be responsible for notifying the solver of the change. When it is `false`, as here, `set_data()` is called with `eModBlck` on both ends instead. The solver also pushes a generic `NBModification` onto the inner block. Crucially, both branches reuse the same persistent inner `UCBlock` across the entire N^2 loop. They also reuse the same attached solver. No block is deserialised, rebuilt, or reconstructed for UC. Only the strength of the change notification changes between the two branches. A small, targeted incremental update is not used here. Instead, the `NBModification` forces the attached solver to resynchronize its entire internal representation of the model, from the abstract model, on every injection. This is what keeps the computation on the correct constraint-generation path for `ECNetworkBlock`.

One further step is required before `compute()` is called. When the `StochasticBlock` applicator is deserialised, each `DataMapping`'s target is resolved as an `AbstractPath`, relative to the file being read. The demand mapping has an empty path: it targets the `UCBlock` itself. Resolving an empty path returns a null pointer. So `uc_run_cssc()` repoints every mapping's target at the actual live inner block before solving. Without this step, the first `set_data()` call would invoke a setter through a null pointer. It would then crash:

Re-binding DataMapping targets to the live inner block

```
auto * stoch = dynamic_cast< StochasticBlock * >(
 srb→get_scenario_applicator() );
if( auto * inner = stoch→get_inner_block() )
 for( const auto & m : stoch→get_data_mappings() )
  m→set_caller_from_reference( inner );
```

This design choice is forced by the limitation of the reused-block incremental path discussed above. Demand reaches the model via `UCBlock::set_active_power_demand`. Its incremental abstract-model update has an out-of-bounds index bug inside `ECNetworkBlock`, when the number of nodes differs from the number of intervals. Reload mode sidesteps this. It forces the persistent solver to resynchronize fully on every scenario injection, via the `NBModification`, rather than trusting the incremental `DataMapping` path. The cost is N^2 full solver resynchronisations during Step 1. This is markedly more expensive than the CFL path. The CFL path only needs a small, targeted incremental update per injection. Reload mode also limits the practical scenario pool size for these experiments (Section 9.4.2).

9.3.5 Scenario Generation

Scenarios are produced by a dedicated standalone tool, `UCScenarioGenerator`, adapted to the structure of a UC instance. It loads the base `ActivePowerDemand` profile (and, when enabled, the `MaxPower` profile of every `IntermittentUnitBlock`) and generates perturbed scenarios organised into three demand clusters: normal (0.9–1.1×), high (1.2–1.4×), and low (0.6–0.8×). When the requested scenario count exceeds the number of base cluster combinations, additional scenarios are produced by applying a small random perturbation (0.95–1.05×) to an existing scenario. An optional validation pass solves each candidate scenario with a time-limited solver and regenerates any infeasible scenario with a milder perturbation, moving to progressively lower-demand clusters if needed. All scenarios are saved to a single `netCDF` file together with uniform probabilities $1/N$. For the EC experiments in this chapter, `UCScenarioGenerator` is run producing demand-only scenarios.

9.4 Experimental Setup

9.4.1 Instances

Experiments are conducted on Energy Community instances. The `EC_CO_Test_TUB` instance used in these experiments has `NumberNodes` = 10 and `NumberIntervals` = 96.

9.4.2 Parameter Ranges

The same two parameters as in the CFL experiments are varied: the number of full scenarios N and the number of representative scenarios K. CSSC Step 1 on UC needs N^2 full solver resynchronisations, through a forced `NBModification`, instead of a small

incremental update. This replaces the N^2 lightweight LP re-solves used for CFL. So the practical range of N is markedly smaller than for CFL. Values of N beyond approximately 30 were observed to exhaust available memory during preliminary runs. This limitation of the current per-cell resynchronisation strategy would need to be addressed to scale further. The specific (N, K) grid used is reported together with the results in Section 9.5.

9.5 Results and Analysis

This section reports preliminary results on the `EC_CO_Test_TUB` instance with demand-only uncertainty, $N = 20$, and $K \in \{5, 10\}$. As discussed in Section 9.4.2, the reload-mode strategy required by `ECNetworkBlock` makes larger values of N impractical at present, so this section covers a single, small-scale configuration. Additional instances and parameter combinations will be added as further experiments complete.

9.5.1 Solution Quality

Table 9.1 reports the in-sample gap for all five methods. The best results of each row are in bold

Table 9.1: Mean in-sample gap (%) on `EC_CO_Test_TUB`, $N = 20$.

K	Baseline	Dupacova	BestFit	FirstFit	CSSC
5	5.176	0.372	0.372	0.372	**0.037**
10	5.185	0.005	0.005	0.005	**0.001**

CSSC achieves the lowest gap at both values of K. At $K = 5$, CSSC's gap of 0.037% is roughly 10× lower than the best heuristic (Dupacova, BestFit, and FirstFit all tie at 0.372%). At $K = 10$, all methods improve sharply, and CSSC's gap drops to 0.0006%, recovering the full-problem objective almost exactly, the best heuristic gap of 0.0049% is still roughly 8× higher. As on the CFL instances, Dupacova, BestFit, and FirstFit again select an identical representative set and produce identical gaps, confirming that the Wasserstein-optimal selection is unique for this scenario set regardless of which of the three selection strategies is used.

Baseline stands out as a clear outlier, with a gap around 5.18% at both $K = 5$ and $K = 10$, showing no improvement as K increases. Its reduced objective is consistently below the full-problem optimum, indicating that Baseline is systematically selecting low-demand scenarios rather than scenarios representative of the full distribution. This suggests that selection by probability weight alone provides little useful signal on EC demand scenarios, where the scenario structure may not be uniform enough for random selection to be informative.

9.5.2 Computational Time

Table 9.2 reports the reduction time (`AlgoTime`) and the reduced TSS solve time (`RedTime`) for all five methods.

Table 9.2: Computational time on `EC_CO_Test_TUB`, $N = 20$.

K	Method	AlgoTime (s)	RedTime (s)	Total (s)
5	Baseline	0.013	0.538	0.551
5	Dupacova	0.014	0.579	0.593
5	BestFit	0.014	0.536	0.551
5	FirstFit	0.014	0.546	0.559
5	CSSC	83.267	0.703	83.970
10	Baseline	0.016	1.379	1.395
10	Dupacova	0.016	1.337	1.353
10	BestFit	0.017	1.401	1.417
10	FirstFit	0.017	1.358	1.375
10	CSSC	107.284	1.548	108.832

The heuristic `AlgoTime` remains negligible, between 13 and 17 milliseconds regardless of K, exactly as observed on the CFL instances. CSSC's `AlgoTime`, by contrast, is 83.3 s at $K = 5$ and 107.3 s at $K = 10$, an overhead of roughly 5,800× and 6,200× relative to the heuristics respectively. This overhead is consistent with the reload-mode strategy described in Section 9.3.4. There are $N^2 = 400$ cells in the opportunity-cost matrix. Each one requires the attached solver to fully resynchronise its representation of the model, via the forced `NBModification`. Each one also requires fixing the commitment variables x_{it}. Each one also requires a MILP or LP solve.

The growth from $K = 5$ to $K = 10$ at fixed $N = 20$ is modest (1.29×), consistent with the CFL observation that `AlgoTime` is dominated by Step 1, whose cost depends on N (the N^2 sub-problem count) rather than on K. The small increase observed here is attributable to Step 2's MILP partitioning, whose size grows with K.

The reduced TSS solve time is comparable across all five methods at the same K, exactly as observed on the CFL instances, confirming that the dominant cost of CSSC lies entirely in the reduction step itself and not in the difficulty of the reduced problem it produces.

Considering the total solution time (`AlgoTime` + `RedTime`) rather than `AlgoTime` alone changes the picture substantially. While CSSC's `AlgoTime` overhead relative to the heuristics is in the order of 5,800–6,200×, the total time overhead drops to roughly 80–150×, since `RedTime` on this UC instance is much larger than on the CFL instances and contributes more evenly across methods. Moreover, as K increases from 5 to 10, the heuristics' total time grows by a factor of 2.5×, driven entirely by the larger reduced problem, while CSSC's total time grows by only 1.3×, since its `AlgoTime` is dominated by Step 1 and depends on N rather than K. This suggests that as K increases, CSSC's relative time disadvantage narrows even further, in addition to its already substantial advantage in solution quality.

9.5.3 Conclusion

These preliminary results confirm that CSSC achieves substantially lower gaps than all heuristic methods on this EC instance, by roughly one order of magnitude at both $K = 5$ and $K = 10$, at the cost of a reduction step that is three to four orders of magnitude slower due to the reload-mode strategy. Baseline is a notable exception among the heuristics, performing significantly worse than the other three methods as discussed above. These results are based on a single instance and a single value of N, a broader assessment across different EC instances and larger N is left for future experiments.

Chapter 10

Computational Experiments on the Capacity Expansion Problem

This chapter presents computational experiments on a two-stage stochastic capacity expansion problem derived from realistic European energy-system data. The instances we used come from Energy team at the University of Pisa, representing a transmission network at the 2050 planning horizon. This setting provides a practical test of whether the gap advantages of CSSC observed on synthetic instances carry over to real-world investment planning.

Section 10.1 formulates the problem. Section 10.2 maps the model onto SMS++ blocks. Section 10.3 describes the two instances. Section 10.4 briefly describes evaluation metrics. Section 10.5 reports and analyses the results.

10.1 The Two-Stage Stochastic Capacity Expansion Problem

The capacity expansion problem asks how much generation, storage, and transmission capacity to invest in now, before uncertain future demand and renewable availability are known, so as to minimize the sum of investment cost and expected operating cost. This is a first-stage / two-stage structure: investment decisions are irreversible and must be committed before the scenario is revealed, while system operation is optimized ex-post for each realized scenario.

The here-and-now investment vector $\mathbf{x} \in \mathbb{R}^{|\mathcal{G}^{inv}|+2|\mathcal{B}^{inv}|+|\mathcal{L}^{des}|}$ must be fixed before the scenario is revealed:

$$x_g^{RE} \geq 0, \quad g \in \mathcal{G}^{inv}, \tag{10.1}$$

$$x_b^{bat} \geq 0, \quad b \in \mathcal{B}^{inv}, \tag{10.2}$$

$$x_b^{conv} \geq 0, \quad b \in \mathcal{B}^{inv}, \tag{10.3}$$

$$z_l \geq 0, \quad l \in \mathcal{L}^{des}, \tag{10.4}$$

where x_g^{RE} is additional installed capacity (MW) of generating unit g, x_b^{bat} and x_b^{conv} are additional storage (MWh) and converter (MW) capacity of storage unit b, and z_l is additional transmission capacity (MW) of line l. The first-stage cost is:

$$c_1(\mathbf{x}) = \sum_{g \in \mathcal{G}^{inv}} c_g^{RE}\, x_g^{RE} + \sum_{b \in \mathcal{B}^{inv}} \left(c_b^{bat}\, x_b^{bat} + c_b^{conv}\, x_b^{conv} \right) + \sum_{l \in \mathcal{L}^{des}} c_l^{net}\, z_l. \tag{10.5}$$

Given $\mathbf{x}$ and a realized scenario ω, the second-stage recourse problem minimizes the total operating cost subject to constraints of generation, storage, network and power balance. Since all second-stage variables are continuous and the generating units carry no binary commitment, the recourse problem is a linear programme. The second-stage cost $Q(\mathbf{x}, \omega)$ depends on $\mathbf{x}$ through the expanded capacities entering the generation limits, storage bounds, and transmission flow limits.

$$\min_{\mathbf{x} \geq 0} \; c_1(\mathbf{x}) + \sum_{\omega=1}^{N} p_\omega\, Q(\mathbf{x},\, \omega). \tag{10.6}$$

With N equiprobable scenarios ($p_\omega = 1/N$), the extensive form is a large-scale LP linking all scenario copies through $\mathbf{x}$ via non-anticipativity constraints. The stochastic optimum v^* is the optimal value of (10.6).

10.2 SMS++ Components

The `UCBlock` hosts four unit types. `IntermittentUnitBlock` models renewables and simplified gas without binary commitment, units with non-zero investment cost contribute x_g^{RE}. `BatteryUnitBlock` models all storage technologies (electrical batteries, hydrogen, CO_2, ammonia, methanol) with a unified charge/discharge model, each investable unit contributes x_b^{bat} and x_b^{conv}. `HydroUnitBlock` models reservoir hydro and pumped storage with no investment variables. `SlackUnitBlock` absorbs nodal imbalance at high penalty to ensure feasibility. The network component is a `DesignNetworkBlock` whose expandable lines each contribute one variable z_l.

10.3 Instances

Two instances are used, both representing a power system at the 2050 planning horizon and converted via `pypsa2smspp`. They share time horizon $T = 24$ and differ in spatial resolution.

Table 10.1: Capacity expansion instances derived from PyPSA networks.

Instance	T	N	**Nodes**	**Lines**	**Interm.**	**Slack**	**Battery**	**Hydro**
`base_s_2`	24	3, 10	15	31	10	2	12	4
`base_s_5`	24	3, 10	121	323	71	49	60	10

`base_s_2` aggregates the grid to 15 nodes and 31 lines with 28 units. The scenario vector has size 792, decomposing as $15 \times 24 = 360$ entries for nodal demand plus $9 \times 24 = 216$ entries each for renewable maximum power and operating cost. `base_s_5` uses a finer 121-node, 323-line representation with 190 units and a scenario vector of size 4 008, and serves to assess how methods scale with network complexity.

Experiments are conducted at two scenario pool sizes: $N = 3$ and $N = 10$. Scenarios are generated by applying scenario-level multipliers sampled from the ranges [0.60, 1.60] for nodal demand, [0.70, 1.20] for renewable maximum power, and [0.50, 1.80] for operating costs, using a fixed random seed to ensure reproducibility. High-demand scenarios are deliberately coupled with reduced renewable availability, so that scenarios differ not only in cost magnitude but in the type of investment they favour. All scenarios carry equal probability $1/N$.

The $N = 3$ configuration provides a minimal but interpretable setting where the V-matrix can be inspected directly and the behaviour of each method traced to individual scenario characteristics. The $N = 10$ configuration extends the comparison to a richer scenario distribution while remaining tractable for the $N^2 = 100$ LP solves required by CSSC.

10.4 Test Framework and Evaluation Metrics

The test program reads the pre-built `TwoStageStochasticBlock` directly and compares all five methods at $K = 1$ representative scenario. The cost-space V-matrix is constructed explicitly: $V[i][j]$ is the total system cost when the investment decision $\mathbf{x}^*(i)$, optimized for scenario i alone, is applied under scenario j. Building V requires N^2 LP solves.

We used two metrics in this chapter. First, the in-sample gap follows the same definition as in the CFL and EC experiments, we described in detail at the section 7.3. Next, we used the implementation error as defined in (3.2). This quantifies the extra cost of applying a single-scenario investment plan across the full distribution. This metric is crucial for irreversible investment decisions.

10.5 Results and Analysis

10.5.1 Solution Quality

Tables 10.2 and 10.3 report the mean gap and implementation error for all five methods across all four configurations tested.

Table 10.2: Results at $K = 1$, $N = 3$.

Instance	Method	Pick	In-sample gap	Impl. error	Time (ms)
base_s_2	Baseline	0	32.19%	54.30%	1
	Dupacova	1	9.45%	5.65%	0
	BestFit	1	9.45%	5.65%	0
	FirstFit	1	9.45%	5.65%	0
	CSSC	2	**1.87%**	**0.38%**	3,181
base_s_5	Baseline	0	35.78%	39.42%	2
	Dupacova	1	7.78%	4.68%	2
	BestFit	1	7.78%	4.68%	1
	FirstFit	1	7.78%	4.68%	1
	CSSC	2	**4.84%**	**3.80%**	35,367

Table 10.3: Results at $K = 1$, $N = 10$.

Instance	Method	Pick	In-sample gap	Impl. error	Time (ms)
base_s_2	Baseline	0	31.01%	48.46%	4
	Dupacova	4	15.99%	10.45%	1
	BestFit	4	15.99%	10.45%	2
	FirstFit	4	15.99%	10.45%	1
	CSSC	8	**0.82%**	**0.62%**	35,005
base_s_5	Baseline	0	32.64%	37.51%	13
	Dupacova	4	14.19%	8.87%	13
	BestFit	4	14.19%	8.87%	14
	FirstFit	4	14.19%	8.87%	12
	CSSC	8	**6.29%**	**6.29%**	400,448

Baseline selects scenario 0 on every instance and pool size, consistently choosing the lowest-demand, highest-renewable scenario. Implementation errors range from 37.51% to 54.30%, confirming that probability-weighted selection provides no useful signal for investment planning when all scenarios are equiprobable: the selected scenario is simply the first one generated, with no regard to cost robustness.

Dupacova, BestFit, and FirstFit produce identical results in every configuration. At $N = 3$ they select scenario 1 (the geometric median of the three-scenario set) and at $N = 10$ they select scenario 4 on both instances. Compared to Baseline, the improvement

is substantial: implementation errors fall to 4.68–5.65% at $N = 3$ and 8.87–10.45% at $N = 10$. However, the error increases from $N = 3$ to $N = 10$ for these methods. A richer scenario pool shifts the distributional centroid further from the stress scenario, moving it away from the cost-robust representative.

CSSC selects scenario 2 at $N = 3$ and scenario 8 at $N = 10$ on both instances, in both cases the high-demand, low-renewable scenario. The selection is driven directly by the V-matrix: the chosen scenario's row is the flattest in expected cost, meaning its investment plan performs well across all realisations. This is most visible in the $N = 10$ V-matrix for `base_s_5`, where row 8 $(15.25, 15.29, \ldots, 15.55, 17.19) \times 10^9$ is nearly flat across the first nine scenarios and only rises for the extreme scenario 9, while row 0 ranges from 9.86×10^9 to 30.5×10^9, a three-fold spread. Refer to table A4 in the Appendix for details.

On `base_s_2`, CSSC's advantage over the distance-based methods is 15× at $N = 3$ and grows to 17× at $N = 10$ (0.38% vs. 5.65%, then 0.62% vs. 10.45%). On `base_s_5`, the advantage is 1.2× at $N = 3$ (3.80% vs. 4.68%) and 1.4× at $N = 10$ (6.29% vs. 8.87%).

The results across all four configurations deliver insights. Distance-based methods consistently select the geometrically central scenario, which is closest to the distribution centroid in Wasserstein distance. In contrast, CSSC consistently selects the cost-robust scenario that minimizes the expected cost over the full distribution. Because these objectives differ, the selected scenarios are generally different. This leads to implementation errors that differ by a factor of 1.2× to 17×, depending on the instance size and the scenario pool size.

The gap between the two approaches widens as N increases on both instances. At $N = 3$, with only three scenarios, the geometric centroid (scen 1) is also a reasonable proxy for the distribution. At $N = 10$, it increases the implementation error of distance-based methods from ∼5% to ∼9–10%. CSSC is unaffected by this, since it operates in cost space rather than scenario space: more scenarios simply means a richer V-matrix from which to identify the robust representative.

This distinction matters most precisely in the setting that motivates scenario reduction for investment planning: irreversible, long-horizon decisions where the cost of underinvestment is asymmetric and large. A 5–54% difference in implementation error, as observed here, translates directly into billions of EUR in a real energy planning exercise.

10.5.2 Computational Time

From the table 10.4, heuristic methods are always minor at all across all tests. For CSSC, three clear patterns stand out regarding its run time. Growth with N on `base_s_2`. From $N = 3$ to $N = 10$, CSSC's total time grows from 3.2 s to 35 s (11×), consistent with the $O(N^2)$ prediction of $(10/3)^2 \approx 11\times$ for Step-1. The individual LP sub-problems remain roughly constant in difficulty as N grows on this smaller instance.

Table 10.4: Computational time at $K = 1$.

Instance	**N**	**Heuristics**	**CSSC Step-1**	**CSSC Step-2**	**CSSC total**	**RedTime**	**Full TSS**
`base_s_2`	3	< 1 ms	1,120 ms	2,061 ms	3,181 ms	∼200 ms	497 ms
`base_s_2`	10	< 5 ms	12,816 ms	22,189 ms	35,005 ms	∼162 ms	1,642 ms
`base_s_5`	3	< 2 ms	11,701 ms	23,666 ms	35,367 ms	∼1,900 ms	47,134 ms
`base_s_5`	10	< 15 ms	94,855 ms	305,593 ms	400,448 ms	∼2,070 ms	697,331 ms

Growth with instance size at $N = 10$. From `base_s_2` to `base_s_5` at $N = 10$, CSSC grows from 35 s to 400 s (11×), driven by the larger LP sub-problems (448 first-stage variables vs. 57, and a 121-node network vs. 15 nodes). Step-2 alone reaches 306 s on `base_s_5` at $N = 10$, dominating the total.

On `base_s_5`, the full three-scenario TSS solve takes 47.1 s ($N = 3$) and 697.3 s ($N = 10$), both longer than CSSC's 35.4 s and 400.4 s respectively. This counter-intuitive result arises because the pre-built `TwoStageStochasticBlock` for `base_s_5` shares first-stage variables across scenario replicas in the SMS++ serialisation, producing degenerate non-anticipativity constraints that significantly slow the LP solver. CSSC avoids this issue entirely, since each of its N^2 sub-problems is a single-scenario LP with no non-anticipativity structure. At larger N, where the full extensive form becomes increasingly intractable, CSSC remains the only practical path to a cost-aware representative selection.

Considering the total time, including `RedTime`, does not change the picture much here. `RedTime` is small and similar across all five methods, typically under 2.3 s even on the larger `base_s_5` instance. Since CSSC's total time is dominated by Step-1 and Step-2, adding `RedTime` changes the overhead ratio between CSSC and the heuristics by less than 1%. This is different from the CFL and UC cases, where `RedTime` was large enough to narrow the gap between CSSC and the heuristics. Here, the implementation error gap is the dominant factor: CSSC is worth its extra cost mainly because of the large reduction in implementation error, not because the running-time overhead shrinks once the reduced problem is solved.

10.5.3 Conclusion

Four findings stand out from these experiments on the capacity expansion instances.

First, the cost of choosing the wrong representative scenario is large and asymmetric. Implementation errors range from 37% to 54% for Baseline and from 5% to 10% for the distance-based methods, compared to 0.4% to 6.3% for CSSC. On a real investment planning exercise at the scale of the transmission system, a 10% implementation error translates directly into billions of EUR of avoidable expenditure.

Second, CSSC consistently identifies the high-demand, low-renewable stress scenario across all four configurations. This scenario produces the most robust investment plan. In contrast, distance-based methods select the geometrically central scenario, which under-builds for the stress case.

Third, CSSC's advantage over the distance-based methods grows as N increases from 3 to 10 (15× to 17× on `base_s_2`, 1.2× to 1.4× on `base_s_5`). This is the opposite of the CFL pattern, where CSSC's advantage narrowed with N. The difference reflects the nature of the problem: in capacity expansion, the cost asymmetry between under- and over-investment is large and persistent, so the cost-space signal exploited by CSSC becomes more valuable as the scenario pool grows and the geometric centroid drifts further from the stress scenario.

Fourth, on `base_s_5`, CSSC is faster than the full extensive-form solve at both $N = 3$ and $N = 10$, highlighting a practical advantage of the N^2 single-scenario approach over direct extensive-form solving when the latter suffers from numerical difficulties at scale.

Chapter 11

Conclusion

This thesis presented an implementation of the Cost-Space Scenario Clustering (CSSC) algorithm within the SMS++ optimization framework. It evaluated this implementation against four distribution-driven heuristics. Two classes of two-stage stochastic programming problems were used for this evaluation.

The main technical contribution is a fully problem-agnostic implementation of CSSC. The starting point was a CFL-specific solver. It was tightly coupled to `CapacitatedFacility LocationBlock`. The final design consists of two independent components. `Scenario ReductionBlock` is a new communication layer. It decouples the scenario data source, the `DiscreteScenarioSet`, from any solver. Both heuristic and CSSC solvers can therefore operate on the same interface. Neither needs any knowledge of the underlying problem type. `CSSCScenarioReductionSolver` is a new solver class. It inherits directly from `Solver`. It has no dependency on any specific block type. Problem-specific behaviour is delegated to two caller-provided callbacks, plus one configuration flag. `VarExtractor` identifies the first-stage decision variables. `DataInjector` propagates scenario data to the attached solver, when direct notification is used. A boolean flag, `f_fix_with_mod`, switches to a reload mode instead. In this mode the solver forces the attached solver to fully resynchronise on every scenario injection, through a generic `NBModification`. This is what makes the same persistent inner block usable even for block types whose incremental data-update path cannot be trusted, such as `ECNetworkBlock`. No fresh block is ever constructed for this purpose. The same solver core can therefore be applied to any two-stage stochastic problem in SMS++. A caller only needs to supply the two callbacks and set this flag appropriately. The solver itself never needs to be modified. The refactored `ScenarioReductionSolver` was also updated. It now reads scenario data directly from `DiscreteScenarioSet`. This removed the previous requirement to construct a synthetic CFLB, solely to encode scenario weights and distances.

The implementation was validated on two problem types. For the Capacitated Facility Location problem, experiments used two OR-Library instances, `cap71` and `cap121`, across $N \in \{20, 50, 100\}$ and $K \in \{5, 10\}$, averaged over 10 seeds. CSSC consistently achieved lower optimality gaps than all four heuristics. The advantage was most pronounced at $N = 20$. There, CSSC recovered the full-problem objective almost exactly at $K = 10$. At larger N, the advantage narrowed. At $N = 100$, $K = 5$, on the easier instance `cap71`, the

heuristics produced a marginally lower gap instead. On the harder instance `cap121`, CSSC retained a clear advantage at $K = 10$, even at $N = 100$. This suggests something about cost-space information: its benefit is more persistent when second-stage costs are more sensitive to the choice of representative. For the Unit Commitment problem, preliminary experiments used one Energy Community instance, at $N = 20$. They confirmed that the generic implementation produces correct results on a structurally different block type. CSSC achieved roughly one order of magnitude lower gaps than the heuristics there too. The implementation was further validated on a real-world capacity expansion problem. The Energy team at the University of Pisa provided transmission network data for the 2050 planning horizon. CSSC achieved implementation errors between 0.4% and 6.3%. The distance-based heuristics reached 5% to 10%. Baseline reached 37% to 54%. On a real transmission system, this difference translates into billions of EUR. Unlike the CFL case, CSSC's advantage over the heuristics grew as N increased, rather than narrowing. This reflects a large and persistent cost asymmetry in this setting, between under-investment and over-investment.

CSSC's reduction time grows faster than the $O(N^2)$ prediction. That prediction is based on sub-problem count alone. From $N = 50$ to $N = 100$, the observed growth was 5–10×. The predicted growth was only 4×. The practical implication is clear. CSSC is most cost-effective when the reduction step is computed once. The representative set can then be reused across multiple downstream solves. For large N, perhaps $N \gg 500$, solving the MILP exactly becomes impractical. Heuristic alternatives should be considered instead in that regime. One open direction for future work is to address this limitation. A scalable clustering heuristic in the cost space could replace the exact MILP, for example.

Several other directions are open for future work. On the scalability side, the unit commitment experiments are currently limited to $N = 20$. The reload-mode mechanism used for `ECNetworkBlock` is responsible for this limit. It reuses the same persistent inner block for every scenario injection. It also forces the attached solver to fully resynchronise on every injection, through a generic `NBModification`. This is far more expensive than the small, targeted incremental update used for CFL. It limits the practical scenario pool size as a result. One natural direction for future work is to fix the known bug in `ECNetworkBlock::set_active_power_demand`. Fixing it would enable the direct-notification, incremental-injection strategy instead. This scalability constraint would then be removed. UC experiments could then be conducted at the same scale as the CFL experiments reported in this thesis.

Finally, all experiments in this thesis consider a single source of uncertainty. This is customer demand in the CFL setting, and user demand in the EC setting. Real-world problems often involve multiple simultaneous uncertainty sources in practice. Power systems, for example, often face joint uncertainty in both demand and renewable generation. Extending the CSSC framework to multi-dimensional scenario spaces is therefore a natural next step.

In conclusion, this thesis presents a step forward in problem-driven scenario reduction for two-stage stochastic optimization. At the same time, it highlights the need for continued research. Scalability, richer uncertainty models, and more rigorous solution evaluation all remain open.

Appendix

Table A1: Cost-space Matrix V for Instance `base_s_2` ($N = 3, K = 1$ values in $\times 10^9$)

	S0	**S1**	**S2**
P0	2.485	6.271	8.207
P1	3.235	3.318	5.061
P2	3.611	3.691	3.733

Table A2: Cost-space Matrix V for Instance `base_s_2` ($N = 10, K = 1$, values in $\times 10^9$)

	S0	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**
P0	2.520	3.347	3.699	4.219	4.984	5.904	6.138	7.132	7.869	8.407
P1	2.701	2.722	3.006	3.499	4.225	5.135	5.369	6.363	7.100	7.638
P2	2.769	2.790	2.799	3.236	3.953	4.847	5.081	6.075	6.812	7.350
P3	2.870	2.889	2.898	2.910	3.564	4.436	4.661	5.655	6.391	6.929
P4	3.012	3.031	3.039	3.051	3.068	3.874	4.084	5.055	5.792	6.330
P5	3.189	3.208	3.215	3.226	3.243	3.263	3.437	4.334	5.052	5.590
P6	3.228	3.247	3.254	3.265	3.281	3.302	3.307	4.183	4.889	5.426
P7	3.422	3.441	3.449	3.460	3.476	3.494	3.499	3.522	4.153	4.645
P8	3.568	3.587	3.595	3.606	3.621	3.640	3.644	3.666	3.682	4.138
P9	3.676	3.695	3.702	3.713	3.729	3.747	3.752	3.773	3.789	3.801

Table A3: Cost-space Matrix V for Instance `base_s_5` ($N = 3, K = 1$, values in $\times 10^9$)

	S0	**S1**	**S2**
P0	9.689	23.479	29.934
P1	13.683	13.912	19.781
P2	15.499	15.663	15.817

Table A4: Cost-space Matrix V for Instance `base_s_5` ($N = 10, K = 1$, values in $\times 10^9$)

	S0	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**
P0	9.855	13.069	14.214	15.976	18.537	21.967	22.391	26.085	28.642	30.455
P1	10.821	10.903	11.990	13.624	16.035	19.398	19.819	23.513	26.070	27.883
P2	11.161	11.208	11.262	12.885	15.233	18.524	18.937	22.626	25.183	26.996
P3	11.671	11.714	11.732	11.797	14.100	17.295	17.693	21.304	23.861	25.674
P4	12.394	12.434	12.450	12.474	12.555	15.657	16.027	19.506	21.994	23.803
P5	13.371	13.410	13.425	13.447	13.480	13.579	13.942	17.283	19.665	21.395
P6	13.477	13.517	13.531	13.553	13.586	13.671	13.692	17.026	19.394	21.109
P7	14.528	14.568	14.582	14.604	14.634	14.677	14.684	14.793	17.099	18.735
P8	15.251	15.291	15.305	15.326	15.356	15.397	15.404	15.453	15.551	17.186
P9	15.762	15.802	15.816	15.838	15.868	15.907	15.913	15.960	16.005	16.088

Reference

1. Keutchayan, J., Ortmann, J., & Rei, W. (2023). Problem-driven scenario clustering in stochastic optimization. *Computational Management Science*, 20(1), 13.

2. Feng Y, & Ryan SM (2016) Solution sensitivity-based scenario reduction for stochastic unit commitment. *Comput Manag Sci* 13(1):29–62

3. Dupačová, J., Gröwe-Kuska, N., & Römisch, W. (2003). Scenario reduction in stochastic programming. *Mathematical Programming*, 95(3), 493–511.

4. Sun M, Teng F, Konstantelos I, & Strbac G (2018) An objective-based scenario selection method for transmission network expansion planning with multivariate stochasticity in load and renewable energy sources. *Energy* 145:871–885

5. Hewitt, M., Ortmann, J., & Rei, W. (2021). Decision-based scenario clustering for decision-making under uncertainty. *Annals of Operations Research*.

6. Bertsimas, D., & Mundru, N. (2022). Optimization-based scenario reduction for data-driven two-stage stochastic optimization. *Operations Research*, 71(4), 1343–1361.

7. Barabino, E., Fioriti, D., Guerrazzi, E., Mariuzzo, I., Poli, D., Raugi, M., Razaei, E., Schito, E., & Thomopulos, D. (2023). Energy communities: A review on trends, energy system modelling, business models, and optimisation objectives. *Sustainable Energy, Grids and Networks*, 36, Article 101187.

8. Fioriti, D., Pampado, A., Scarpelli, C., Pasini, G., Lutzemberger, G., & Barsali, S. (2025). Enhancing energy system modelling with advanced mathematical decomposition techniques: Feasibility of coupling SMS++ and PyPSA. In *Proceedings of the 2025 IEEE International Conference on Environment and Electrical Engineering (EEEIC / I&CPS Europe)*. IEEE.

9. Frangioni, A., et al. SMS++: Structured Modeling System in C++. University of Pisa. Available at: `https://gitlab.com/smspp/smspp-project`