FROM SEQUENTIAL TO PARALLEL: REFORMULATING DYNAMIC PROGRAMMING AS GPU KERNELS FOR LARGE-SCALE STOCHASTIC COMBINATORIAL OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Dynamic programming (DP) is central to combinatorial optimization, optimal control, and reinforcement learning, yet its perceived sequentiality has long hindered scalability. We introduce a general-purpose GPU framework that reformulates broad classes of forward DP recursions as batched min–plus matrix–vector products over layered DAGs, collapsing actions into masked state-to-state transitions that map directly to GPU kernels. This approach removes a major bottleneck in scenario-based stochastic programming (SP), where the use of DP has traditionally restricted the number of scenarios due to excessive computational cost. Our framework exposes massive parallelism across scenarios, transition layers, and, when applicable, route or action options, via self-designed GPU kernels that implement Bellman updates with warp-/block-level reductions and numerically safe masking. In a single GPU pass, these kernels can process over 10^6 uncertainty realizations, far beyond the capacity of prior scenario-based methods. We demonstrate the approach in two canonical SP applications: (i) a vectorized split operator for the capacitated vehicle routing problem with stochastic demand, exploiting *2D* parallelism (scenarios $\times$ transitions); and (ii) a forward inventory reinsertion DP under an order-up-to policy, exploiting *3D* parallelism (scenarios $\times$ inventory transitions $\times$ route options). Across benchmarks, the implementation scales nearly linearly in the number of scenarios and achieves one to three orders of magnitude speedups over multithreaded CPU baselines, yielding tighter SAA estimates and consistently stronger first-stage decisions under identical wall-clock budgets. Viewed as hardware-aware software primitives, our min–plus DP kernels offer a drop-in path to scalable, GPU-accelerated stochastic discrete optimization.

1 INTRODUCTION

Dynamic programming (DP) is a foundational paradigm in optimization and control, enabling the decomposition of complex multi-stage decision problems into tractable subproblems. Classic references such as Bertsekas' monograph *Dynamic Programming and Optimal Control* (Bertsekas, 2012) have established DP as a unifying framework for sequential decision making. Through Bellman's principle of optimality, DP characterizes value functions and optimal policies recursively. This dual role - as both an analytical tool and a computational method - has made DP central to fields such as operations research, control, reinforcement learning, and artificial intelligence.

DP plays a particularly important role in combinatorial optimization. On the one hand, it serves as the foundation of several exact algorithms, such as the Held–Karp procedure for the traveling salesman problem (TSP, Held & Karp, 1971), pseudo-polynomial algorithms for knapsack-type problems (Shapiro, 1968), and classical shortest path algorithms (e.g., Bellman, 1958; Dijkstra, 1959). On the other hand, DP often appears as a subproblem solver within more complex frameworks. For instance, in vehicle and inventory routing problems (VRP, Prins, 2004; Vidal, 2016), DP-based "split" algorithms efficiently decompose giant tours into feasible routes. This decomposition is a cornerstone in many state-of-the-art heuristics and metaheuristics, as it enables the rapid evaluation of candidate solutions and thus directly impacts overall algorithmic performance. Moreover, column

generation approaches typically rely on DP to solve resource-constrained shortest path problems in the pricing step, often through label-setting algorithms (Feillet et al., 2004; Irnich & Desaulniers, 2005). These algorithms are not only theoretically elegant but also practically indispensable, as they form one of the most widely adopted subroutines in VRP, crew scheduling, job-shop scheduling and other huge integer programs (IP, Barnhart et al., 1998). Beyond these direct applications, metaheuristic methods, which are widely used to tackle complex real-world NP-hard problems, often exploit DP to efficiently evaluate and repair candidate solutions (Vidal et al., 2012; Zhao et al., 2022), thereby enabling the exploration of large and sophisticated neighborhoods without incurring prohibitive computational costs. As exact and meta-heuristic methods represent the two principal approaches to solving combinatorial optimization problems (CO), DP is thus not merely a standalone solver; it also serves as a versatile backbone embedded in many of the most powerful methods in this domain.

Moving beyond deterministic settings, stochastic optimization constitutes another major branch of combinatorial optimization, typically formulated as integer or mixed-integer programs that incorporate uncertainty to better capture the dynamics of real-world systems. In stochastic optimization, DP is the canonical lens through which multistage decision making is modeled under uncertainty (Bertsekas, 2012). Classical stochastic DP provides the foundation for exact formulations, while approximation methods, such as such as approximate dynamic programming (Powell, 2007), neuro-dynamic programming (Bertsekas & Tsitsiklis, 1995), and reinforcement learning (Staddon, 2020). Moreover, many decomposition frameworks embed DP-based subroutines to evaluate recourse actions across scenarios (Birge & Louveaux, 1997; Shapiro et al., 2021).

A key observation is that stochastic programming (SP) naturally lends itself to parallelism: once the first-stage decision is fixed, second-stage recourse evaluations across scenarios are independent. Building on this structure, we develop a GPU-accelerated framework that reformulates DP algorithms for stochastic combinatorial optimization into fully vectorized, GPU-parallelizable routines. This framework enables the simultaneous evaluation of millions of uncertainty realizations in a single GPU pass, significantly improving both runtime and statistical accuracy in sample-average approximation (SAA).

Our work introduces the first scenario-based solver capable of handling over one million realizations in stochastic combinatorial optimization, substantially extending the scalability frontier of scenario-based methods. While GPU-based acceleration has been applied to continuous optimization (e.g., GPU-ADMM (Schubiger, 2019), CuPDLP (Lu & Yang, 2023), GPU-QP (Bishop et al., 2024), GPU-IPM (Liu et al., 2024)), no comparable approach has been developed for discrete, combinatorial, and scenario-based stochastic problems. We demonstrate the effectiveness of the framework on two canonical applications: the capacitated vehicle routing problem with stochastic demand (CVRPSD) and the dynamic stochastic inventory routing problem (DSIRP). In both cases, DP subroutines that were previously sequential are reformulated in matrix form and implemented via efficient GPU kernels, achieving performance levels previously unattainable. In summary, our main contributions are as follows.

1. We show that many forward dynamic programming recursions in stochastic combinatorial problems can be rewritten as batched min–plus matrix–vector products over layered DAGs. This algebraic view collapses actions into masked state-to-state transitions that map directly to GPU kernels.
2. We develop GPU kernels that exploit both scenario-level and transition-level parallelism: two-dimensional parallelism for the VRP split operator, and three-dimensional parallelism for the inventory reinsertion operator. Bellman minimizations are implemented as warp-/block-level GPU reductions, with padding and masking ensuring numerical stability.
3. A single GPU pass evaluates more than 10^6 uncertainty realizations, yielding near-linear scaling in the number of scenarios and one to three orders of magnitude acceleration over multithreaded CPU baselines.
4. This throughput translates directly into decision quality: tighter sample-average approximations and stronger first-stage solutions within the same wall-clock budget.
5. Finally, we distill a general recipe—state layering, transition masking, min–plus batching, and reductions—that can turn a broad class of DP subroutines into high-throughput GPU primitives for stochastic discrete optimization.

Overall, the paper demonstrates that classic DP subroutines, once perceived as inherently sequential, can be systematically re-expressed in min-plus matrix form and executed as high-throughput GPU kernels, unlocking modern hardware for large-scale stochastic discrete optimization. Our implementation is available at `https://github.com/Jingyi-poly/2-stage-IRP-GPU`.

2 Generic Dynamic Programming Framework

Preliminary Knowledge. We consider a finite-horizon dynamic program over stages $t = 1, \dots, T$, starting from an initial state $s_1 \in \mathcal{S}_1$. At each stage t, the system is in a state $s_t \in \mathcal{S}_t$, an action $a_t \in \mathcal{A}_t(s_t)$ is chosen, and the system moves to s_{t+1} either deterministically via $g_t(s_t, a_t)$ or stochastically according to $P_t(s_{t+1} \mid s_t, a_t; \omega)$. Each transition incurs a stage cost $c_t(s_t, a_t; \omega)$ under scenario ω.

Let $J_t^\omega(s)$ denote the minimum cumulative cost to reach $s \in \mathcal{S}_t$ at stage t. The recursion initializes as $J_1^\omega(s_1) = 0$ and evolves by

$$J_{t+1}^\omega(s') = \min_{\substack{s \in \mathcal{S}_t,\, a \in \mathcal{A}_t(s) \\ g_t(s,a) = s'}} \{J_t^\omega(s) + c_t^\omega(s, a)\}, \quad \forall s' \in \mathcal{S}_{t+1}. \tag{1}$$

The objective is to minimize the expected terminal cost $\mathbb{E}_\omega\big[\min_{s \in \mathcal{S}_T} J_T^\omega(s)\big]$.

2.1 Transition-Based DAG Formulation.

To enable GPU-friendly computation, we reformulate the recursion using state-to-state transitions. Define the forward transition cost matrix:

$$A_t^\omega(s, s') := \inf_{\substack{a \in \mathcal{A}_t(s) \\ g_t(s,a) = s'}} c_t^\omega(s, a), \quad \text{with } A_t^\omega(s, s') = +\infty \text{ if infeasible.}$$

Then the forward update becomes:

$$J_{t+1}^\omega(s') = \min_{s \in \mathcal{S}_t} \{J_t^\omega(s) + A_t^\omega(s, s')\}. \tag{2}$$

This recursion corresponds to a layered shortest-path expansion over a forward DAG, where transitions are edges from layer $\mathcal{S}_t$ to layer $\mathcal{S}_{t+1}$.

2.2 Min-Plus Matrix Formulation.

Let $\mathcal{S}_t = \{1, \dots, m_t\}$ index the state space. Define:

$$A_t^\omega(i, j) = A_t^\omega(s = i,\, s' = j), \qquad J_t^\omega \in \mathbb{R}^{m_t}.$$

Then, the Bellman update becomes a matrix-vector product in the $(\min, +)$ semiring:

$$J_{t+1}^\omega = (A_t^\omega)^\top \otimes J_t^\omega := \Big[\min_i \{A_t^\omega(i, j) + J_t^\omega(i)\}\Big]_{j=1}^{m_{t+1}}. \tag{3}$$

This min-plus formulation enables efficient GPU implementation via tensor broadcasting and dimension-wise minimization, with infeasible transitions masked via $+\infty$. For variable-sized state spaces, padding and masking ensure regular tensor shapes for parallel execution. A demonstrative DP example (5.1), together with the full formulations of the following two applications, VRPSD split (5.2) and DSIRP reinsertion (5.3), is provided in the Appendix.

2.3 Instantiation A: Split DP on a Giant Tour in the Vehicle Routing Problem with Stochastic Demand.

Problem Motivation. In vehicle routing postprocessing, a common task is to split a "giant tour" $\sigma = [\sigma_1, \dots, \sigma_n]$ into capacity-feasible routes. Given demands $q_{\sigma_k}^\omega$ under scenario ω and vehicle capacity Q, define state i as having served customers σ_1 to σ_i. An action $p < i$ ends the previous route at p, starting a new one from $p{+}1$ to i. The departure depot is denoted by 0, and the destination depot by $n{+}1$. The DP explores all possible cut points $p < i$ that define where to start a new route, and accumulates the minimal total travel cost for serving customers up to i. (see the Figure 7 in 5.2 for better understanding).

Forward DP Recursion and Matrix Form. The cost of serving subroute $[\sigma_{p+1}, \ldots, \sigma_i]$ is

$$W^\omega(p,i) \;=\; c_{0,\sigma_{p+1}} \;+\; \sum_{k=p+1}^{i-1} c_{\sigma_k,\sigma_{k+1}} \;+\; c_{\sigma_i,n+1},$$

which is feasible only if $\sum_{k=p+1}^{i} q^\omega_{\sigma_k} \le Q$. We define masked transition entries as

$$A^\omega(p,i) \;=\; \begin{cases} W^\omega(p,i), & \text{if } p<i \text{ and } \sum_{k=p+1}^{i} q^\omega_{\sigma_k} \le Q, \\ +\infty, & \text{if } p<i \text{ and } \sum_{k=p+1}^{i} q^\omega_{\sigma_k} > Q \quad \text{(capacity violated)}, \\ +\infty, & \text{if } p \ge i \quad \text{(not applicable)}. \end{cases}$$

Let $J^\omega(0)=0$ and $J^\omega(i)$ be the optimal cost to reach state i. The forward-DP update is then

$$J^\omega(i) \;=\; \min_{p<i}\{J^\omega(p) + A^\omega(p,i)\}, \qquad i=1,\ldots,n,$$

which is equivalently expressed as the masked min–plus reduction

$$J^\omega(i) \;=\; \big(A^\omega(\cdot,i)\big)^\top \otimes J^\omega(0\!:\!i\!-\!1), \tag{4}$$

where $\otimes$ denotes the $(\min,+)$ semiring product and $A^\omega(\cdot,i)$ is the i-th column with all infeasible or undefined entries masked by $+\infty$ (see Appendix 5.2 for a numerical toy example).

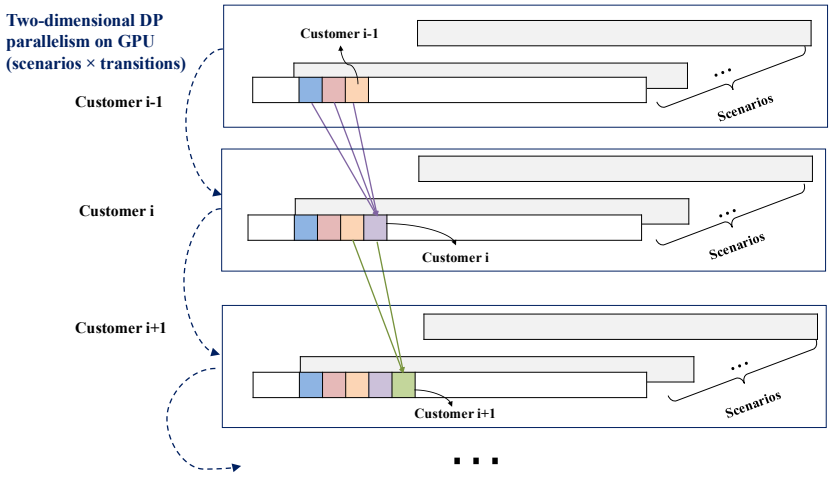


Figure 1: Two-dimensional DP parallelism on GPU (scenarios × predecessors). Each row corresponds to a destination state i, each block within a row represents a scenario ω, and colored bars indicate feasible predecessors $p < i$. For each (i,ω) pair, all predecessors are expanded in parallel to form the set $\{J^\omega(p) + A^\omega(p,i) : p<i\}$, followed by a column-wise min-reduction over p that yields $J^\omega(i)$.

Two-Dimensional Parallelism on GPU. From equation 4, the computation at a fixed destination state i factorizes over the Cartesian product $\Omega \times \{p : p<i\}$. We therefore exploit two-dimensional GPU parallelism across **scenarios** ω and **predecessors** $p<i$. Each thread computes one pair (ω,p) by loading $A^\omega(p,i)$ and the partial cost $J^\omega(p)$, forming $J^\omega(p)+A^\omega(p,i)$. A warp-/block-level *min* reduction across p then yields $J^\omega(i)$ for that scenario. Launching such kernels for all scenarios in parallel computes the masked min–plus reduction of $A^\omega(\cdot,i)$ against $J^\omega(0:i\!-\!1)$. As illustrated in Figure 1, this structure maps naturally to GPUs: scenarios ω are parallelized across columns, predecessors p are reduced within blocks, and rows (states i) advance independently along the DP frontier.

2.4 Instantiation B: Forward Inventory Reinsertion DP in Dynamic Stochastic Inventory Routing Problems.

Problem Motivation. In the stochastic inventory routing problem, delivery schedules are often determined at an aggregate level and then refined through local reinsertion moves: a customer i that

risks a stockout is reconsidered, and new visits are inserted into existing routes or additional trips are scheduled. The key challenge is that reinsertion must balance two competing effects: (i) earlier deliveries create higher holding cost and may cause inefficiency in vehicle loading, (ii) postponing deliveries increases the probability of future stockouts under adverse demand realizations. The goal is to decide, for each customer, *when* to replenish under uncertain demand so as to minimize expected routing, holding, and stockout costs. DP provides a natural way to resolve this trade-off, as it captures the temporal coupling of inventory states and demand uncertainty.

The decision process follows a two-stage stochastic optimization framework. In the first stage, a delivery and routing plan is established for Day 1. Specifically, the model determines which customers to replenish and how much to deliver, subject to vehicle capacity and routing constraints. These decisions are made prior to the realization of demand and are identical across all demand scenarios. Once the demand over the entire planning horizon (Days 1 to H) is realized, second-stage decisions are made adaptively from Day 2 onward. These include scenario-dependent routing and replenishment actions that respond to the realized demand in each scenario (see the Figure 8 in 5.2 for better understanding).

The objective is to *minimize the total expected cost*, which consists of two components: (1) First-stage costs including the Day 1 routing cost and the supplier's inventory holding cost; and (2) Second-stage costs, which vary across scenarios and include: inventory holding costs and stock-out penalties at customers at the end of Day 1; routing and delivery costs from Day 2 to H; and inventory holding and stock-out penalties from Day 2 to H.

The DP operator used to solve this problem follows the formulation in (Zhao et al., 2025), which focuses on a single customer i over a planning horizon $t = 1, \ldots, H$, starting with initial inventory I_i^0 and capacity U_i. The customer's demand is uncertain and modeled by a finite set of scenarios Ω, where $d_i^{t,\omega}$ denotes demand on day t under scenario ω. For each customer, the decision at each day consists of:

- whether customer i is visited at time t,
- the delivery quantity q_i^t, which can only take two values: $q_i^t \in \{0,\ U_i - I_i^{t-1}\}$ that is, either no delivery or replenishment up to full capacity.
- and which vehicle route is chosen to accommodate this visit.

These decisions are scenario-independent, i.e., the same schedule applies across all $\omega \in \Omega$ while inventory evolution is scenario-dependent. The state variable is the end-of-day inventory $I_i^{t,\omega}$, updated as

$$I_i^{t,\omega} = \max\{0,\ I_i^{t-1,\omega} + q_i^t - d_i^{t,\omega}\}, \quad \forall t,\ \omega.$$

Here the DP systematically evaluates both replenishment options (no delivery vs full OU delivery), propagating inventory states forward in time and accumulating costs. See Appendix 5.3 for a numerical toy example.

Forward DP Recursion and Matrix Form. At each day t, customer i either receives no delivery ($q_i^t = 0$) or is replenished up to capacity ($q_i^t = U_i - I_i^{t-1,\omega}$). The per-stage cost for scenario ω consists of two components: (1) routing and detour costs associated with sending q_i^t, denoted $F_t(q_i^t)$; and (2) customer-side inventory holding and stock-out penalties $h_i^t(I_i^{t,\omega})$, evaluated at the end-of-day inventory $I_i^{t,\omega}$.

Let $C_i^t(I_i^{t,\omega})$ denote the minimum expected cumulative cost up to day t for customer i under scenario ω, given that the day-t starting inventory is $I_i^{t,\omega}$. By construction, $C_i^t(\cdot)$ is a piecewise linear function of the inventory state. The forward recursion is

$$C_i^{t+1}(I_i^{t+1,\omega}) \ = \min_{q_i^t \in \{0,\, U_i - I_i^{t,\omega}\}} \Big\{ C_i^t(I_i^{t,\omega}) + F_t(q_i^t) + h_i^t(I_i^{t+1,\omega}) \Big\},$$

where the inventory state evolves as $I_i^{t+1,\omega} = \max\{0,\ I_i^{t,\omega} + q_i^t - d_i^{t,\omega}\}$. The recursion starts from the initial inventory before day 1: $C_i^0(I_i^{0,\omega}) = 0$ with $I_i^{0,\omega} = I_i^0$.

To enable GPU-friendly computation, we collapse the action space into a state-to-state transition matrix:

$$A_i^{t,\omega}(I,J) := \min_{\substack{q\in\{0,U_i-I\} \\ \max\{0,I+q-d_i^{t,\omega}\}=J}} \big\{F_t(q)+h_i^t(J)\big\}, \quad +\infty \text{ if no feasible } q \text{ leads from } I \text{ to } J.$$

In practice, evaluating each entry $A_i^{t,\omega}(I,J)$ may itself require enumerating a finite set of candidate route options (e.g., alternative reinsertion choices), in which case

$$A_i^{t,\omega}(I,J) = \min_{r\in\mathcal{K}} A_i^{t,\omega}(I,J;r).$$

Rows correspond to today's starting inventory I, columns to tomorrow's inventory J, and each entry stores the minimal cost of transitioning from I to J.

Define the state space

$$\mathcal{S}_i^t := \{I \in \mathbb{Z}_{\geq 0} : 0 \leq I \leq U_i\},$$

and collect $C_i^t(I)$ over $I \in \mathcal{S}_i^t$ as a vector $J_i^t \in \mathbb{R}^{|\mathcal{S}_i^t|}$. The forward recursion then becomes a masked min–plus matrix–vector product:

$$J_i^{t+1} = (A_i^{t,\omega})^\top \otimes J_i^t := \Big[\min_{I\in\mathcal{S}_i^t}\big\{A_i^{t,\omega}(I,J)+J_i^t(I)\big\}\Big]_{J\in\mathcal{S}_i^{t+1}}. \tag{5}$$

Three-Dimensional Parallelism on GPU. From the matrix form, the computation at stage t factorizes over the Cartesian product $\Omega \times \{(I \to J)\} \times \mathcal{R}$, where $\mathcal{R}$ denotes the set of candidate route options for each transition. We therefore exploit three-dimensional GPU parallelism across **scenarios** ω, **state transitions** $I \to J$, and **route options** $r \in \mathcal{R}$. Each thread computes one tuple $(\omega, I{\to}J, r)$ by loading $A_i^{t,\omega}(I,J;r)$ and the partial cost $J_i^t(I)$, forming $J_i^t(I) + A_i^{t,\omega}(I,J;r)$. A warp-/block-level *min* reduction is first performed across route options r, then across predecessor states I, yielding $J_i^{t+1}(J)$ for each scenario ω. Launching such kernels for all ω in parallel realizes the batched column-wise min–plus updates of the recursion, while also vectorizing over alternative delivery routes.

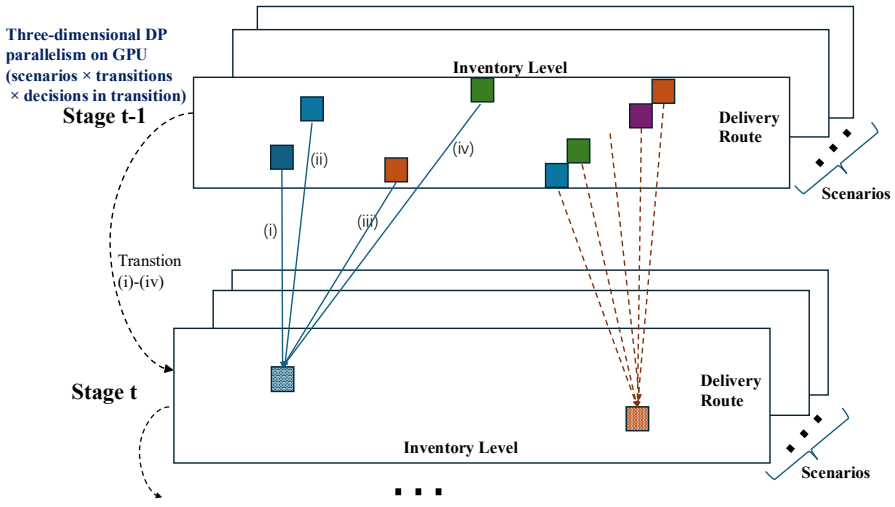


Figure 2: Three-dimensional DP parallelism on GPU (scenarios × transitions × route options). Each layer corresponds to a stage t, with nodes representing end-of-day inventory levels I. Colored edges denote feasible transitions $I \to J$ under scenario-specific demands $d^{t,\omega}$. For each tuple $(\omega, I{\to}J, r)$, threads evaluate the cost contribution $J_i^t(I) + A_i^{t,\omega}(I,J;r)$, combining routing overhead with holding and stockout penalties. A two-level reduction (first across route options r, then across predecessor states I) yields $J_i^{t+1}(J)$ per scenario. The figure highlights how GPU parallelism spans scenarios, transitions, and route options, turning the DP recursion into a fully batched min–plus update.

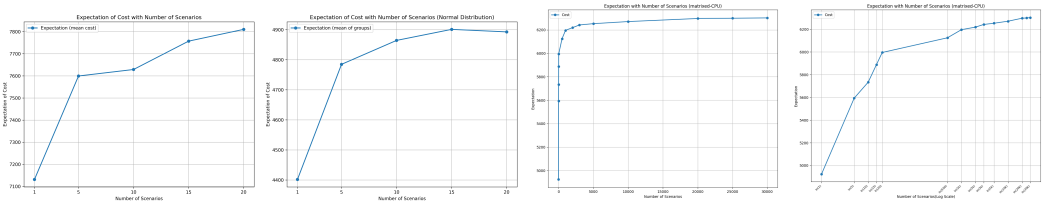
3 EXPERIMENTS

3.1 SCALING THE SCENARIO SIZE IN STOCHASTIC PROGRAMMING.

The theoretical properties of empirical risk minimization (ERM) under mild regularity conditions establish that SAA solutions may suffer from *bias* with small sample sizes but converge *consistently* toward the true optimum as the number of scenarios grows, with an asymptotic $\mathcal{O}(1/\sqrt{m})$ convergence rate (see Appendix 5.4 for a formal statement). To examine how these properties manifest in practice, we conduct experiments on the DSIRP, a representative setting where demand distributions are complex and cannot be adequately captured by simple parametric families. Figure 3 reports the empirical behavior of SAA estimators as the number of scenarios increases.

Specifically, Figures 3a and 3b highlight the bias effect: with only a small number of scenarios, the estimated cost systematically underestimates the true expectation. As the sample size grows, the estimator increases and gradually stabilizes, consistent with the theoretical consistency guarantee. Figures 3c and 3d further illustrate the convergence behavior. As the number of scenarios increases from hundreds to tens of thousands, the SAA estimate approaches the true optimum, and its variance decreases at the predicted $\mathcal{O}(1/\sqrt{m})$ rate. The log-scaled plot shows that improvements continue to accrue even at large sample sizes, underscoring that "more scenarios" consistently yield better estimates rather than reaching a premature plateau.

In general, these findings demonstrate that when the underlying distribution of uncertainty is complex or unknown, as is common in real-world, data-driven settings, restricting the scenario set to only a few hundred or a few thousand is insufficient. Substantially larger scenario sets are needed to reduce bias and improve solution quality. Our GPU-accelerated DP framework makes such large-scale, data-driven stochastic evaluation computationally feasible in practice.



(a) Bias under random demand distribution (b) Bias under normal demand distribution (c) Convergence with number of scenarios (linear scale) (d) Convergence with number of scenarios (log scale)

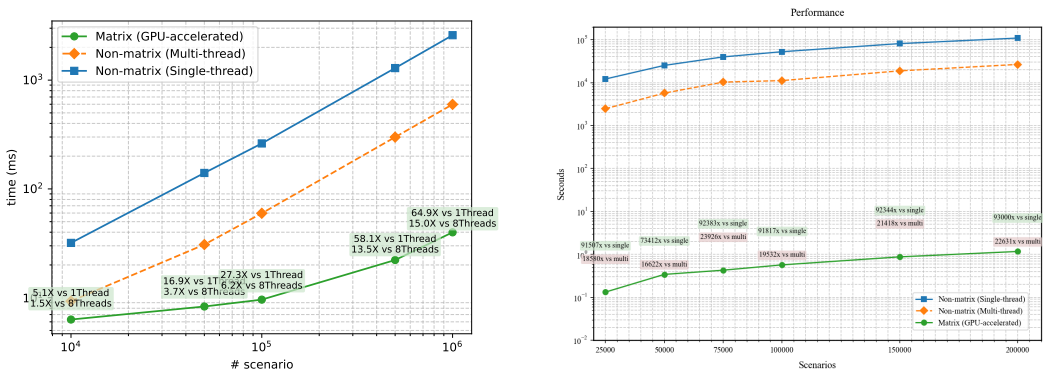
Figure 3: Empirical behavior of SAA estimators in DSIRP. Top: bias under different demand distributions. Bottom: convergence with increasing scenario size.

3.2 SCALABILITY WITH THE NUMBER OF SCENARIOS.

The above results confirm that larger scenario sets are statistically necessary to reduce bias and achieve consistency in stochastic programming. We next examine whether such scaling is computationally feasible. We evaluate the efficiency of our GPU-accelerated DP operators against CPU baselines on two representative tasks: (i) the split operator in VRPSD and (ii) the reinsertion operator in DSIRP. All implementations were written in C++/CUDA and tested on a machine with an AMD Ryzen 7 9700X CPU (8 cores) and an NVIDIA RTX 2080Ti GPU with 11 GB memory. The CPU baselines include (i) a single-threaded implementation and (ii) a multi-threaded implementation with 8 threads. The GPU implementations exploit the two- (or three-) dimensional parallelism described in Section 2.

Left (CVRPSD split DP). As the number of scenarios increases from 10^4 to 10^6, the single-thread CPU runtime rises sharply and reaches minutes at 10^6 scenarios, while the 8-thread baseline shows only moderate relief before saturating due to synchronization and memory-bandwidth limits. The GPU curve grows nearly linearly and remains in the seconds range even at 10^6 scenarios, yielding about $80\times$ speedup over single-thread CPU and $20\times$ over the 8-thread baseline at the largest setting.

Right (DSIRP reinsertion DP). The effect is far more dramatic. At 2×10^5 scenarios, the GPU implementation attains roughly 9.3×10^4 speedup versus the single-thread CPU and 2.26×10^4 versus the multi-thread CPU (see callouts in the figure). These gains stem from: (i) three-dimensional



(a) VRPSD split operator

(b) DSIRP reinsertion operator

Figure 4: Runtime comparisons of CPU and GPU implementations across different scaling regimes. Left: scaling behavior under 10^4–10^6 scenarios for VRPSD. Right: large-scale evaluation for DSIRP.

parallelism (scenarios $\times$ inventory transitions $\times$ route options), (ii) high arithmetic intensity with coalesced memory access, and (iii) warp-/block-level reductions that keep the Bellman minima on-chip.

Across both problems, GPU parallelization shifts the computational frontier for scenario-based evaluation: near-linear scaling in the number of scenarios with order-of-magnitude to five-orders-of-magnitude speedups (problem-dependent). This throughput is what enables the very large, data-driven scenario sets used in our SAA experiments, directly supporting the statistical benefits documented in the previous subsection.

3.3 Impact of Training Scenario Set Size on Decision Quality

We next examine how the number of evaluated scenarios influences the quality of the first-stage decision. Specifically, we solve the problem under different scenario counts, ranging from 1 to 10^4 (i.e., $1, 100, 1{,}000$). For each scenario setting, the obtained first-stage solution is evaluated on a fixed large out-of-sample test set of 10^6 scenarios. Figure 5 reports the out-of-sample cost achieved by the best observed solution on two CVRPSD instances: x-n128 and x-n105.

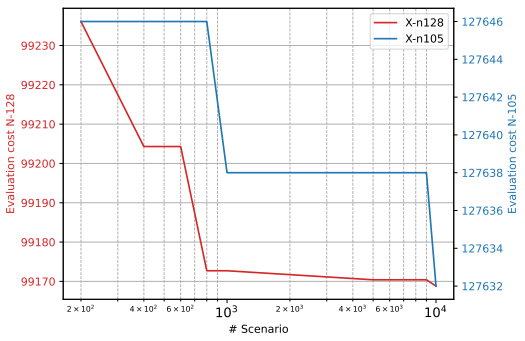


Figure 5: Out-of-sample performance of first-stage solutions obtained with varying observed scenario settings. Larger evaluation set yield more robust and lower-cost solutions.

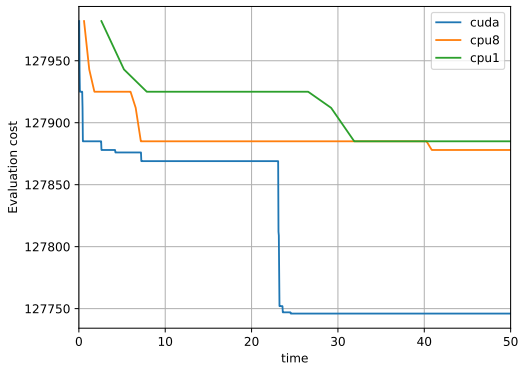


Figure 6: Quality of the best solution obtained at each time under a fixed time budget. GPU consistently achieves better decisions due to faster evaluation and thus larger effective search effort.

Results. When observing only few scenarios (e.g., a single scenario), the resulting first-stage solution is severely biased and performs poorly. Increasing the scale of available scenarios consistently improves robustness, with significant gains observed throughout Figure 5. Such performance

confirms the theoretical insight that larger sample sizes reduce estimation bias in sample-average approximation. These results demonstrate that our GPU-based framework, by enabling the evaluation of tens or hundreds of thousands of scenarios within practical runtimes, leads to significantly more reliable first-stage solutions compared to CPU-based methods that are restricted to only a few thousand scenarios.

3.4 Decision Quality under Fixed Time Budgets

Finally, we compare the decision quality obtained under identical wall-clock time limits across the three implementations: CPU single-thread, CPU multi-thread, and GPU. Each method is given a fixed runtime budget , during which the split algorithm splits giant tours to obtain first-stage solutions. For fairness, all approaches are evaluated on the same problem instance with 10^4 available scenarios. Figure 6 reports the best penalized cost obtained within the allowed runtime.

Results. With small time budgets, all methods return feasible but suboptimal solutions, yet GPU already provides a noticeable advantage. As the time limit increases, the quality gap widens: GPU produces solutions that are consistently closer to the true optimum, while CPU single-thread stagnates and CPU multi-thread improves only modestly. This matches intuition: faster scenario evaluation allows the GPU to explore many more candidate first-stage tours within the same runtime, thereby improving the probability of discovering high-quality solutions.

These results confirm that beyond scalability, GPU acceleration directly translates into superior decision quality under realistic time constraints, making it particularly valuable in operational settings where decisions must be made quickly.

4 Conclusion

We showed that forward dynamic programs commonly used in stochastic combinatorial optimization can be reformulated as batched min–plus matrix–vector products over layered DAGs, collapsing actions into masked state-to-state transitions that map directly to GPU kernels. This algebraic view exposes 2D/3D parallelism across scenarios, transitions, and, when applicable, route or action options, enabling warp-/block-level Bellman reductions with numerically safe masking. On two representative applications—a vectorized split operator for CVRPSD and a forward reinsertion DP for DSIRP—our implementation scales nearly linearly in the number of scenarios and achieves one to three orders of magnitude speedups over multithreaded CPU baselines. The resulting throughput materially improves sample-average approximation quality and yields consistently stronger first-stage decisions under identical wall-clock budgets. Despite these advances, the approach inherits GPU memory constraints, struggles with highly irregular state spaces, and is currently limited to forward DP with additive costs. Future extensions include handling constrained and risk-aware objectives, integrating learned surrogates while preserving numerical safety, applying kernel fusion and multi-GPU scaling, and generalizing beyond DP to other semiring-based dynamic programs and column generation subroutines. Viewed as hardware-aware software primitives, our min–plus DP kernels offer a drop-in path to scalable, GPU-accelerated scenario-based optimization.

References

Cynthia Barnhart, Ellis L Johnson, George L Nemhauser, Martin WP Savelsbergh, and Pamela H Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations research*, 46(3):316–329, 1998.

Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

Dimitri Bertsekas. *Dynamic programming and optimal control: Volume I*, volume 4. Athena scientific, 2012.

Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Proceedings of 1995 34th IEEE conference on decision and control*, volume 1, pp. 560–564. IEEE, 1995.

John R Birge and Francois Louveaux. *Introduction to stochastic programming*. Springer, 1997.

Arun L Bishop, John Z Zhang, Swaminathan Gurumurthy, Kevin Tracy, and Zachary Manchester. Relu-qp: A gpu-accelerated quadratic programming solver for model-predictive control. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 13285–13292. IEEE, 2024.

Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1: 269–271, 1959. doi: 10.1007/BF01386390.

Dominique Feillet, Pierre Dejax, Michel Gendreau, and Cyrille Gueguen. An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems. *Networks: An International Journal*, 44(3):216–229, 2004.

Michael Held and Richard M Karp. The traveling-salesman problem and minimum spanning trees: Part ii. *Mathematical programming*, 1(1):6–25, 1971.

Stefan Irnich and Guy Desaulniers. Shortest path problems with resource constraints. In *Column generation*, pp. 33–65. Springer, 2005.

Felix Liu, Albin Fredriksson, and Stefano Markidis. A gpu-accelerated interior point method for radiation therapy optimization. *arXiv preprint arXiv:2405.03584*, 2024.

Haihao Lu and Jinwen Yang. cupdlp. jl: A gpu implementation of restarted primal-dual hybrid gradient for linear programming in julia. *arXiv preprint arXiv:2311.12180*, 2023.

Warren B Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality*, volume 703. John Wiley & Sons, 2007.

Christian Prins. A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers & operations research*, 31(12):1985–2002, 2004.

Michel Schubiger. Gpu acceleration of admm for large-scale convex optimization. Master's thesis, ETH Zurich, 2019.

Alexander Shapiro. Monte carlo sampling methods. *Handbooks in operations research and management science*, 10:353–425, 2003.

Alexander Shapiro, Darinka Dentcheva, and Andrzej Ruszczynski. *Lectures on stochastic programming: modeling and theory*. SIAM, 2021.

Jeremy F Shapiro. Dynamic programming algorithms for the integer programming problem—i: The integer programming problem viewed as a knapsack type problem. *Operations Research*, 16(1): 103–121, 1968.

JER Staddon. The dynamics of behavior: Review of sutton and barto: Reinforcement learning: An introduction . *Journal of the Experimental Analysis of Behavior*, 113(2), 2020.

T. Vidal, T.G. Crainic, M. Gendreau, N. Lahrichi, and W. Rei. A hybrid genetic algorithm for multidepot and periodic vehicle routing problems. *Operations Research*, 60(3):611–624, 2012.

Thibaut Vidal. Split algorithm in o (n) for the capacitated vehicle routing problem. *Computers & Operations Research*, 69:40–47, 2016.

J. Zhao, M. Poon, Z. Zhang, and R. Gu. Adaptive large neighborhood search for the time-dependent profitable dial-a-ride problem. *Computers & Operations Research*, 147:105938, 2022.

Jingyi Zhao, Claudia Archetti, Tuan Anh Pham, and Thibaut Vidal. Large neighborhood and hybrid genetic search for inventory routing problems. *arXiv preprint arXiv:2506.03172*, 2025.

5 APPENDIX

5.1 ILLUSTRATIVE EXAMPLE FOR THE MIN-PLUS MATRIX-VECTOR BELLMAN UPDATE.

To illustrate the min-plus matrix-vector Bellman update, consider a dynamic programming recursion over three states in stage t and two states in stage $t+1$.

Let the cost-to-go vector at stage t under scenario ω be:

$$J_t^\omega = \begin{bmatrix} 0 \\ 1 \\ 3 \end{bmatrix},$$

and the transition cost matrix from stage t to $t+1$ be:

$$A_t^\omega = \begin{bmatrix} 2 & 5 \\ 1 & +\infty \\ 3 & 0 \end{bmatrix},$$

where $A_t^\omega(i,j)$ denotes the cost of transitioning from state i at stage t to state j at stage $t+1$. The value $+\infty$ denotes an infeasible transition.

Then, the Bellman update in the $(\min, +)$ semiring becomes:

$$J_{t+1}^\omega = (A_t^\omega)^\top \otimes J_t^\omega,$$

which is computed entry-wise as:

$$J_{t+1}^\omega(1) = \min_i \{A_t^\omega(i,1) + J_t^\omega(i)\} = \min\{2+0,\ 1+1,\ 3+3\} = \min\{2,2,6\} = 2,$$

$$J_{t+1}^\omega(2) = \min_i \{A_t^\omega(i,2) + J_t^\omega(i)\} = \min\{5+0,\ +\infty+1,\ 0+3\} = \min\{5,+\infty,3\} = 3.$$

Thus, the updated cost-to-go vector at stage $t+1$ is:

$$J_{t+1}^\omega = \begin{bmatrix} 2 \\ 3 \end{bmatrix}.$$

This computation reflects a forward shortest-path propagation over a layered graph, where the cost of reaching each state in the next stage is determined by minimizing over all incoming transitions from the previous stage.

5.2 INSTANTIATION A: SPLIT DP ON A GIANT TOUR IN THE VEHICLE ROUTING PROBLEM WITH STOCHASTIC DEMAND.

Problem Setting. The stochastic programming community has extensively studied scenario-based formulations, where uncertainty is modeled by a finite set of realizations. However, scenario-based evaluation quickly becomes computationally prohibitive on CPUs, where even tens of thousands of scenarios can overwhelm multi-threaded implementations. In our work, we adopt this *scenario-based modeling* framework, in which customer demands are represented by sampled realizations. This approach naturally accommodates correlated demand structures and supports data-driven modeling when historical records are available. This scenario-based approach falls naturally into the two-stage stochastic programming paradigm, whose general form is: $\min_{x \in X} f_1(x) + \mathbb{E}_\xi [f_2(x,\xi)]$, where x denotes the first-stage routing decisions and the corresponding cost $f_1(x)$, ξ is a random vector representing a realization of customer demands, and $f_2(x,\xi)$ denotes the second-stage recourse cost under each scenario ξ.

In our two-stage stochastic optimization problem, the first stage determines the visiting sequence of customers, commonly referred to as a *giant tour* in the context of genetic algorithms (Vidal et al., 2012). This representation encodes a solution as a permutation of customers, from which feasible vehicle routes can be recovered via a split operator under fixed vehicle capacity and scenario-dependent customer demands. We assume *full demand revelation prior to the second stage*, enabling the plan to adapt to the realized scenario. Once demands are known, the giant tour is *split into feasible routes* such that the demand on each route does not exceed vehicle capacity. Thus, given a fixed giant tour (i.e., the visiting sequence), the second-stage evaluation is computationally simple: it only requires splitting the tour according to realized demands. **The objective is to determine a first-stage giant tour that minimizes the expected total travel cost across all scenarios.**

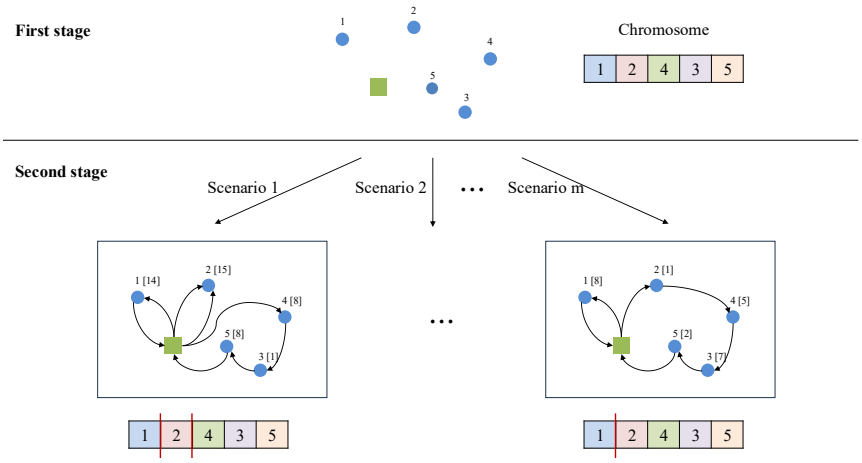


Figure 7: Example of splitting a giant tour into feasible routes under a given demand scenario.

Illustrative Example of the problem. Suppose the first-stage giant tour is $(1, 2, 4, 3, 5)$ with vehicle capacity $Q = 17$, as illustrated in Figure 7.

(i) In Scenario 1, the realized demands are $[14, 15, 8, 1, 8]$. The second-stage routing may then split the tour into three feasible routes: (1), (2), and $(4, 3, 5)$.

(ii) In Scenario m, the realized demands are $[8, 1, 5, 7, 2]$ with three vehicles available. The second-stage routing may then split the tour into two feasible routes: (1) and $(2, 4, 3, 5)$.

It is well understood that training robust first-stage policies benefits from incorporating a large number of demand scenarios to capture the full breadth of uncertainty (Shapiro (2003), see the details in next section. Theoretical results show that a sufficiently large scenario set is critical for achieving accurate and stable estimations. However, in practice, evaluating even tens of thousands of scenarios can be prohibitively expensive, especially for combinatorial problems such as the CVRPSD, since each scenario requires solving a non-trivial routing evaluation (typically through a dynamic programming algorithm with time complexity $O(nP)$, where n is the number of customers and P is the number of possible transitions) rather than a simple function call.

Pseudo-code for 2D Kernel. In this section, we describe the GPU kernel designed to generate data splits for each scenario in Algorithm 2. In Algorithm 1, we launch this kernel with a grid of thread blocks to materialize the output matrix Splits for all scenarios are produced concurrently, which enables simultaneous split generation across all scenarios.

Algorithm 1 GPU Splitting Algorithm

Input: Scenarios $\omega \in \Omega$
Input: Global settings Q, C for vehicle capacity and travel cost matrix.
Output: Running costs V
1: Initialize $V \in \mathbb{R}^{n \times |\Omega|}$ to all zero matrix
2: Instantiate multiple 2D kernels in parallel for V^{ω}.
3: $V = [V^{\omega_0}, V^{\omega_1}, ..., V^{\omega_{|\Omega|}}]$
4: **return** V

Illustrative Matrix Form of DP. To clarify the structure of the transition cost matrix A^{ω} in the split problem, consider a simple instance with a giant tour $\sigma = [\sigma_1, \sigma_2, \sigma_3]$, and realized demands under scenario ω given by:

$$(q^{\omega}_{\sigma_1}, q^{\omega}_{\sigma_2}, q^{\omega}_{\sigma_3}) = (2, 3, 4), \quad \text{with vehicle capacity } Q = 5.$$

Algorithm 2 2D Kernel for Splitting

Input: Scenario Index ω
Input: Scenario-specific demand q^ω
Input: Global settings Q, C for vehicle capacity and travel cost matrix.
Output: Running costs V^ω

```
Initialize V^ω to all zero vectors
Initialize ζ^ω to an empty queue.
for each customer i do
    Update potential V^ω_(i)
    if i < n then
        if i dominates ζ^ω_back then
            while ζ^ω ≠ ∅ AND ζ^ω_back dominates i from right do
                Pop ζ^ω from back
            end while
            Push i to ζ^ω from back
        end if
        while |ζ^ω| > 1 AND ζ^ω_front better than ζ^ω_next_front do
            Pop ζ^ω from front
        end while
    end if
end for
return V^ω
```

Let $f^\omega(i)$ denote the minimum cost to serve customers σ_1 through σ_i. We construct a transition cost matrix $A^\omega \in \mathbb{R}^{3\times 3}$ where the (p, i) entry represents the cost of serving customers σ_{p+1} to σ_i in one route, if the cumulative demand is within capacity.

Assume the travel cost matrix is:

$$[c_{a,b}] = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 1 & 2 \\ 2 & 1 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix},$$

where node 0 is the depot and node 4 is the return depot. Then we compute:

- $A^\omega(0,1)$: route = $[\sigma_1]$, demand = $2 \le 5$: feasible. Cost = $c_{0,1} + c_{1,4} = 1 + 3 = 4$.
- $A^\omega(0,2)$: route = $[\sigma_1, \sigma_2]$, demand = 5 : feasible. Cost = $1 + 1 + 2 = 4$.
- $A^\omega(0,3)$: route = $[\sigma_1, \sigma_2, \sigma_3]$, demand = 9 : infeasible. Cost = $+\infty$.
- $A^\omega(1,2)$: route = $[\sigma_2]$, demand = 3 : feasible. Cost = $c_{0,2} + c_{2,4} = 2 + 2 = 4$.
- $A^\omega(1,3)$: route = $[\sigma_2, \sigma_3]$, demand = 7 : infeasible. Cost = $+\infty$.
- $A^\omega(2,3)$: route = $[\sigma_3]$, demand = 4 : feasible. Cost = $c_{0,3} + c_{3,4} = 3 + 1 = 4$.

Thus, the masked transition matrix becomes:

$$A^\omega = \begin{bmatrix} \mathrm{x} & 4 & 4 & +\infty \\ \mathrm{x} & \mathrm{x} & 4 & +\infty \\ \mathrm{x} & \mathrm{x} & \mathrm{x} & 4 \\ \mathrm{x} & \mathrm{x} & \mathrm{x} & \mathrm{x} \end{bmatrix},$$

the forward DP proceeds *column by column* from the initialization $J^\omega(0) = 0$:

$$\boxed{J^\omega(1) = J^\omega(0) + A^\omega(0,1) = 0 + 4 = 4}$$

(since only the predecessor $p = 0$ is admissible when $j = 1$). Then

$$\boxed{J^\omega(2) = \min\left\{J^\omega(1) + A^\omega(1,2),\ J^\omega(0) + A^\omega(0,2)\right\} = \min\{4+4,\ 0+4\} = 4},$$

and

$$J^\omega(3) = \min\left\{J^\omega(2) + A^\omega(2,3),\ J^\omega(1) + A^\omega(1,3),\ J^\omega(0) + A^\omega(0,3)\right\} = \min\{4+4,\ 4++\infty,\ 0++\infty\} = 8.$$

Thus the running costs after this pass are

$$V^\omega = \big(J^\omega(0), J^\omega(1), J^\omega(2), J^\omega(3)\big) = (0,\ 4,\ 4,\ 8).$$

This is exactly the left-to-right scheme you described: each new $J^\omega(j)$ is the minimum over the *previous* prefixes, i.e., $J^\omega(j) = \min_{r<j}\{J^\omega(r) + A^\omega(r,j)\}$, so the recursion naturally continues as $j \to j+1$ on larger instances.

5.3 Evaluation on Instantiation B: Forward Inventory Reinsertion DP in Dynamic Stochastic Inventory Routing Problems.

Problem Statement: Two-Stage Stochastic Inventory Routing Problem with OU Policy. We consider a two-stage stochastic version of the Inventory Routing Problem (IRP) under an Order-Up-To (OU) inventory policy. We define the model on a complete directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{A})$, where $\mathcal{N} = \{0, n+1\} \cup \mathcal{N}'$ includes the supplier node 0, the destination depot $n+1$, and the set of customers $\mathcal{N}'$. The arc set $\mathcal{A}$ represents all possible directed connections between nodes. A set of vehicles $\mathcal{K}$ is available for deliveries, with $|\mathcal{K}| = K$. The planning horizon spans a finite set of days $\mathcal{T} = \{1, \ldots, H\}$ and $\mathcal{T}' = \{2, \ldots, H\}$ for the second stage. To model uncertainty in demand, we consider a finite set of scenarios Ω, where each scenario $\omega \in \Omega$ occurs with probability p^ω. Each vehicle has a capacity limit Q, and each customer $i \in \mathcal{N}'$ has an inventory capacity U_i. Holding costs h_i are incurred per unit of inventory stored at node i, and a stock-out penalty is incurred at a rate of ρh_i per unit of unmet demand at customer i, where $\rho > 1$. The customer demand at customer i on day t under scenario ω is denoted by $d_i^{t,\omega}$. The cost of traveling from node i to node j is denoted c_{ij} for each arc $(i,j) \in \mathcal{A}$. Finally, I_i^0 represents the initial inventory level at node i at the beginning of the planning horizon.

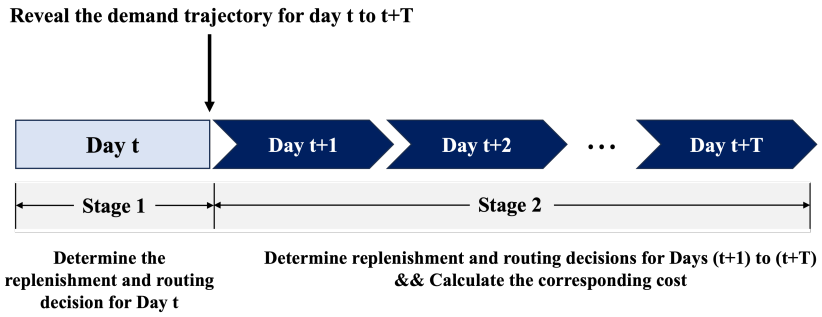


Figure 8: The decision process follows a two-stage stochastic optimization framework. In the **first stage**, a delivery and routing plan is established for Day 1. Specifically, the model determines which customers to replenish and how much to deliver, subject to vehicle capacity and routing constraints. These decisions are made *prior to* the realization of demand and are shared across all possible demand scenarios. Once the demand over the entire planning horizon (Days 1 to H) is realized, **second-stage decisions** are made adaptively from Day 2 onward. These include scenario-specific vehicle routing and replenishment actions that respond to the realized demand in each scenario. The objective is to *minimize the total expected cost*, which consists of two components: (1) First-stage costs, including the Day 1 routing cost and the supplier's inventory holding cost; (2) Second-stage costs, which vary across scenarios and include: (i) Inventory holding costs and stock-out penalties at customers at the end of Day 1; (ii) Routing and delivery costs from Day 2 to H; (iii)Inventory holding and stock-out penalties from Day 2 to H.

To simplify replenishment decisions and reflect common logistics practices, we adopt an OU policy. Under this policy, each customer is either replenished up to its full capacity U_i, or not replenished at all on a given day. This is modeled using binary variables z_i^t (or $z_i^{t,\omega}$ in the second stage), which equal 1 if customer i is replenished on day t, and 0 otherwise.

Master Problem: First-Stage Optimization with Expected Future Cost: The first-stage of the 2SIRP involves determining the vehicle routing and delivery quantities on Day 1, before the actual demand

realizations are observed. Specifically, the first-stage decisions include the quantity $q_i^{k,1}$ delivered to customer i by vehicle k, as well as the routing and loading variables $z_i^{k,1},\ y_{ij}^{k,1}, Q_i^{k,1}$ that describe the path and vehicle load throughout the tour. We use $x = \{q_i^{k,1}, z_i^{k,1},\ y_{ij}^{k,1}, Q_i^{k,1}\}$ to denote all the variables in this stage. Based on these decisions, we formulate the master problem to minimize the total cost on Day 1, including the supplier's inventory holding cost and the routing cost, along with the expected second-stage cost-to-go $\tilde{Q}^{\omega}(x)$ across all demand scenarios $\omega \in \Omega$. The probability of scenario $\omega \in \Omega$ occurring is p^{ω} and $\sum_{\omega\in\Omega} p^{\omega} = 1$.

$$\min_{x} \quad h_0 I_0^1 + \sum_{k\in\mathcal{K}} \sum_{(i,j)\in\mathcal{A}} c_{ij} y_{ij}^{k,1} + \sum_{\omega\in\Omega} p^{\omega} \tilde{Q}^{\omega}(x) \tag{6}$$

$$\text{s.t.} \quad I_0^1 = I_0^0 + d_0^1 - \sum_{k\in\mathcal{K}} \sum_{i\in\mathcal{N}'} q_i^{k,1} \tag{7}$$

$$q_i^{k,1} = (U_i - I_i^0) \cdot z_i^{k,1} \qquad \forall i \in \mathcal{N}',\ \forall k \in \mathcal{K} \tag{8}$$

$$\sum_{i\in\mathcal{N}'} q_i^{k,1} \le Q \cdot z_0^{k,1} \qquad \forall k \in \mathcal{K} \tag{9}$$

$$\sum_{j\in\mathcal{N}'\cup\{n+1\}} y_{ij}^{k,1} = z_i^{k,1}, \quad \sum_{j\in\mathcal{N}'\cup\{0\}} y_{ji}^{k,1} = z_i^{k,1} \qquad \forall i \in \mathcal{N}',\ \forall k \in \mathcal{K} \tag{10}$$

$$\sum_{j\in\mathcal{N}'\cup\{n+1\}} y_{0j}^{k,1} = 1, \quad \sum_{j\in\mathcal{N}'\cup\{0\}} y_{j,n+1}^{k,1} = 1 \qquad \forall k \in \mathcal{K} \tag{11}$$

$$Q_0^{k,1} = \sum_{i\in\mathcal{N}'} q_i^{k,1} \qquad \forall k \in \mathcal{K} \tag{12}$$

$$Q_i^{k,1} - q_i^{k,1} \ge Q_j^{k,1} - Q \cdot (1 - y_{ij}^{k,1}) \qquad \forall (i,j) \in \mathcal{A},\ \forall k \in \mathcal{K} \tag{13}$$

$$q_i^{k,1},\ Q_i^{k,1},\ I_0^1 \ge 0 \qquad \forall i \in \mathcal{N}',\ \forall k \in \mathcal{K} \tag{14}$$

$$z_i^{k,1},\ y_{ij}^{k,1} \in \{0,1\} \qquad \forall i \in \mathcal{N},\ (i,j) \in \mathcal{A},\ \forall k \in \mathcal{K} \tag{15}$$

Constraint equation 7 tracks the inventory at the supplier after Day 1. Constraint equation 8 implements the OU policy: if a customer is visited by a vehicle, the vehicle must deliver exactly enough to fill the inventory up to its capacity U_i; otherwise, no delivery is made. Constraint equation 9 limits total delivery by vehicle capacity. Constraints equation 10–equation 11 ensure routing feasibility. Constraint equation 12 defines the starting load of each vehicle. Constraint equation 13 eliminates subtours. Finally, constraints equation 14–equation 15 define the variable domains.

Cost-to-Go Function and Second-Stage Problem Under Scenario ω: Given the first-stage decision and realized demand $\boldsymbol{d}^{\omega} = \{d_i^{t,\omega}\}_{i\in\mathcal{N},t\in\mathcal{T}'}$ under scenario ω, the second-stage cost-to-go function $\tilde{Q}^{\omega}(x)$ can be calculated as: To ease our exposition, we will use the shorthand notations

$$\boldsymbol{z}^{\omega} = \{z_i^{k,t,\omega}\}_{i\in\mathcal{N},k\in\mathcal{K},t\in\mathcal{T}'} \qquad \boldsymbol{y}^{\omega} = \{y_{ij}^{k,t,\omega}\}_{(i,j)\in\mathcal{A},k\in\mathcal{K},t\in\mathcal{T}'} \tag{16}$$

$$\boldsymbol{q}^{\omega} = \{q_{i,\omega}^{k,t}\}_{i\in\mathcal{N}',k\in\mathcal{K},t\in\mathcal{T}'} \qquad \boldsymbol{I}^{\omega} = \{I_{i,\omega}^{t}\}_{i\in\mathcal{N},t\in\mathcal{T}'} \qquad \boldsymbol{B}^{\omega} = \{B_{i,\omega}^{t}\}_{i\in\mathcal{N}',t\in\mathcal{T}'} \tag{17}$$

and use $\boldsymbol{\xi^{\omega}} = \{\boldsymbol{z^{\omega}}, \boldsymbol{y^{\omega}}, \boldsymbol{q}^{\omega}, \boldsymbol{I}^{\omega}, \boldsymbol{B}^{\omega}\}$, $\omega \in \Omega$ to denote all the variables in the second stage. The objective for the second stage can be represented by

$$\tilde{Q}^{\omega}(x) = \sum_{i\in\mathcal{N}'} \left(h_i I_i^{1,\omega} + \rho h_i B_i^{1,\omega}\right) + \min_{\boldsymbol{\xi^{\omega}}} \sum_{t=2}^{H} \left[h_0 I_0^{t,\omega} + \sum_{i\in\mathcal{N}'} \left(h_i I_i^{t,\omega} + \rho h_i B_i^{t,\omega}\right) + \sum_{k\in\mathcal{K}} \sum_{(i,j)\in\mathcal{A}} c_{ij} y_{ij}^{k,t,\omega}\right]$$

$$\textbf{s.t.} \quad I_i^{1,\omega} = I_i^0 + \sum_{k\in\mathcal{K}} q_i^{k,1} - d_i^{1,\omega} + B_i^{1,\omega} \qquad \forall i \in \mathcal{N}' \tag{18}$$

$$I_i^{t,\omega} = I_i^{t-1,\omega} + \sum_{k\in\mathcal{K}} q_i^{k,t,\omega} - d_i^{t,\omega} + B_i^{t,\omega} \qquad \forall i \in \mathcal{N}',\ t \in \mathcal{T}', \omega \in \Omega \tag{19}$$

$$\sum_{k\in\mathcal{K}} q_i^{k,t,\omega} = (U_i - I_i^{t-1,\omega}) \cdot z_i^{t,\omega} \qquad \forall i \in \mathcal{N}',\ t \in \mathcal{T}', \omega \in \Omega \tag{20}$$

$$\sum_{i \in \mathcal{N}'} q_i^{k,t,\omega} \leq Q \cdot z_0^{k,t,\omega} \qquad \forall k \in \mathcal{K},\ t \in \mathcal{T}', \omega \in \Omega \tag{21}$$

$$\sum_{j \in \mathcal{N}' \cup n+1} y_{ij}^{k,t,\omega} = z_i^{t,\omega}, \quad \sum_{j \in \mathcal{N}' \cup 0} y_{ji}^{k,t,\omega} = z_i^{t,\omega} \qquad \forall i \in \mathcal{N}',\ k \in \mathcal{K},\ t \in \mathcal{T}', \omega \in \Omega \tag{22}$$

$$\sum_{j \in \mathcal{N}' \cup n+1} y_{0j}^{k,t,\omega} = 1, \quad \sum_{j \in \mathcal{N}' \cup 0} y_{j,n+1}^{k,t,\omega} = 1 \qquad \forall k \in \mathcal{K},\ t \in \mathcal{T}', \omega \in \Omega \tag{23}$$

$$Q_0^{k,t,\omega} = \sum_{i \in \mathcal{N}'} q_i^{k,t,\omega} \qquad \forall k \in \mathcal{K},\ t \in \mathcal{T}', \omega \in \Omega \tag{24}$$

$$Q_i^{k,t,\omega} - q_i^{k,t,\omega} \geq Q_j^{k,t,\omega} - Q(1 - y_{ij}^{k,t,\omega}) \qquad \forall (i,j) \in \mathcal{A},\ k \in \mathcal{K},\ t \in \mathcal{T}', \omega \in \Omega \tag{25}$$

$$I_i^{t,\omega},\ B_i^{t,\omega},\ q_i^{k,t,\omega},\ Q_i^{k,t,\omega} \geq 0 \qquad \forall i \in \mathcal{N}',\ k \in \mathcal{K},\ t \in \mathcal{T}', \omega \in \Omega \tag{26}$$

$$z_i^{t,\omega},\ y_{ij}^{k,t,\omega} \in \{0,1\} \qquad \forall i \in \mathcal{N},\ (i,j) \in \mathcal{A},\ k \in \mathcal{K},\ t \in \mathcal{T}', \omega \in \Omega \tag{27}$$

The second-stage constraints under the OU policy govern the evolution of inventory, replenishment quantities, and routing decisions from Day 2 to Day H under each demand scenario $\omega \in \Omega$. Constraint equation 18 sets the initial inventory level for each customer $i \in \mathcal{N}'$ at the end of Day 1, based on the first-stage deliveries, realized demand $d_i^{1,\omega}$, and any unmet demand carried as backorders $B_i^{1,\omega}$. Constraint equation 19 recursively updates the inventory and backorders for subsequent days $t \geq 2$ using the delivery quantities and realized daily demands. Constraint equation 20 enforces the OU policy: a customer either receives a shipment that raises its inventory up to the full capacity U_i (i.e., $q_i^{k,t,\omega} = U_i - I_i^{t-1,\omega}$), or receives nothing. The binary variable $z_i^{t,\omega}$ indicates whether a replenishment occurs at node i on day t. Constraint equation 21 ensures that the total quantity delivered by any vehicle k on day t does not exceed its capacity Q. Constraints equation 22 maintain routing feasibility: if a customer is visited ($z_i^{t,\omega} = 1$), then exactly one arc must enter and one must leave it for each vehicle k. Constraint equation 23 ensures that each vehicle departs from the origin depot and ends at the destination depot once per day. Constraint equation 24 defines the load carried by each vehicle upon departure, equal to the sum of deliveries assigned to it. Constraint equation 25 eliminates subtours by imposing consistency in vehicle loads across successive arcs in a tour. Finally, constraints equation 26–equation 27 impose the appropriate domain restrictions, ensuring non-negativity of inventory, deliveries, and vehicle loads, and binary decisions for vehicle routing and customer visits.

Illustrative Example of the problem. The proposed framework is modular: while the outer layer can be any heuristic or metaheuristic search procedure that explores alternative first-stage decisions (e.g., different delivery plans or routing structures), the inner DP routine ensures that inventory evolution and replenishment actions are evaluated consistently across time and scenarios. Figure 9 illustrates how DP propagates costs forward by updating inventory states and associated routing paths under the OU policy. The blue bars denote customer inventory states, the network diagrams represent feasible routing actions, and the arrows indicate transitions with their corresponding costs. Infeasible transitions are marked as $+\infty$. By capturing the before-and-after evolution of replenishment paths, the DP routine provides a structured way to evaluate candidate solutions, regardless of the outer search framework in which it is embedded.

Pseudo-code for 3D Kernel. To efficiently adapt the CPU-based DP algorithm for GPUs, we design a two-part framework. Specifically, Algorithm 3 selects a GPU-feasible batch size, solves scenarios in batches via a DP kernel, and aggregates results, reducing the batch size and retrying upon out-of-memory. Algorithm 4 executes a batched backward dynamic program over the horizon—parallel across scenarios and states—comparing "no delivery" versus "deliver to capacity," recording the minimizer and transition, and then backtracking from each scenario's initial inventory to recover daily decisions, quantities, and total cost.

Illustrative Matrix Form of DP. Consider a single customer with $U_i = 2$, horizon $t = 1, 2$, and a single scenario ω. The initial inventory is $I_i^0 = 1$, daily demand is $d_i^{t,\omega} = 1$, the transportation cost

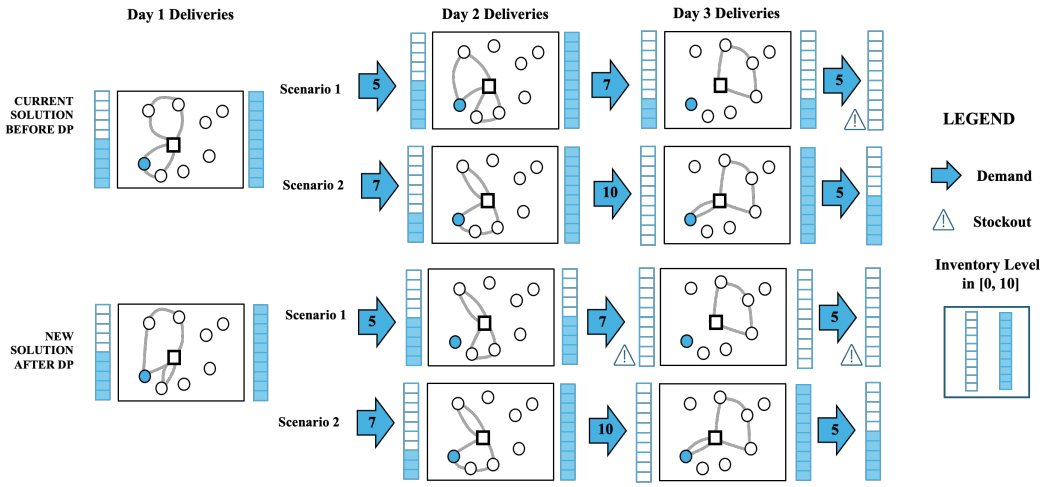


Figure 9: Illustration of the dynamic programming (DP) routine under the Order-Up-To (OU) policy. Blue bars represent possible inventory states, network diagrams show routing decisions, and arrows indicate state transitions with their associated costs. Infeasible transitions are marked as $+\infty$. The DP systematically propagates costs forward, capturing the before-and-after evolution of replenishment paths; this routine can be embedded within any heuristic or metaheuristic framework.

Algorithm 3 GPU-Based DP with Adaptive Batching

Input: `params`, `client_id`, `all_scenarios_data`, initial batch size B_0, device
Output: List of scenario results (cost, decision flags, quantities)

$B \leftarrow$ AdjustBatchSize(B_0, device, `params`) ▷ based on free GPU memory
`results` $\leftarrow [\,]$; `start` $\leftarrow 0$; $N \leftarrow$ #scenarios
while `start` $< N$ **do**
  `batch` $\leftarrow$ slice(`all_scenarios_data`, `start`:`start`+B)
  (`costs`, `flags`, `qty`) $\leftarrow$ SolveBatchDP(`batch`, `params`, `device`)
  Append per-scenario tuples to `results`
  `start` $\leftarrow$ `start` $+B$
  ClearCache()
end while
return `results`

Algorithm 4 SolveBatchDP

function SolveBatchDP(`batch`, `params`, `device`)
  Preprocess tensors (time-major, contiguous, to device); set horizon T, state grid $\mathcal{S}$
  Initialize DP arrays C, D, P; set $C[T] \leftarrow 0$
  for $t = T-1, \ldots, 0$ **do** ▷ (parallel over scenarios/states on GPU)
    Compute no-delivery cost and next state
    Compute deliver-to-max cost (fixed + capacity + holding) and next state
    $C[t] \leftarrow \min(\cdot)$; $D[t] \leftarrow$ argmin decision; $P[t] \leftarrow$ next-state index
  end for
  return Backtrack(C, D, P, start inventories) ▷ costs, flags, quantities
end function

is $F_t(q) = q$, and the end-of-day inventory cost is

$$h_i^t(I) = \begin{cases} 0, & I = 2, \\ 1, & I = 1, \\ 5, & I = 0. \end{cases}$$

The state space each day is $\mathcal{S}_i^t = \{0, 1, 2\}$.

Day 1. The forward transition matrix $A_i^{1,\omega}(I, J)$ (rows I, columns J) is

$$A_i^{1,\omega} = \begin{bmatrix} 5 & 3 & +\infty \\ 5 & 2 & +\infty \\ +\infty & 1 & +\infty \end{bmatrix}, \quad (\ q \in \{0,\, U_i - I\},\ J = \max\{0, I + q - 1\}\ ).$$

The initial cost vector encodes $I_i^0 = 1$:

$$J_i^1 = \begin{bmatrix} +\infty \\ 0 \\ +\infty \end{bmatrix}.$$

The column-wise min–plus update gives

$$J_i^2(J) = \min_{I \in \{0,1,2\}} \{J_i^1(I) + A_i^{1,\omega}(I, J)\}, \quad J_i^2 = \begin{bmatrix} \min\{\infty{+}5,\ 0{+}5,\ \infty\} \\ \min\{\infty{+}3,\ 0{+}2,\ \infty{+}1\} \\ \min\{\infty,\ \infty,\ \infty\} \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \\ +\infty \end{bmatrix}.$$

Day 2. Parameters are the same, so $A_i^{2,\omega} = A_i^{1,\omega}$. Updating once more:

$$J_i^3(J) = \min_{I} \{J_i^2(I) + A_i^{2,\omega}(I, J)\}, \quad J_i^3 = \begin{bmatrix} \min\{5{+}5,\ 2{+}5,\ \infty\} \\ \min\{5{+}3,\ 2{+}2,\ \infty{+}1\} \\ \min\{\infty,\ \infty,\ \infty\} \end{bmatrix} = \begin{bmatrix} 7 \\ 4 \\ +\infty \end{bmatrix}.$$

With a two-day horizon, the terminal cost is $\min_J J_i^3(J) = 4$.

Policy insight. - Day 1: from $I = 1$, delivering $q = 1$ (replenish to full) is optimal, leading to $J = 1$ with cost $1 + 1 = 2$. - Day 2: again from $I = 1$, delivering $q = 1$ is optimal, adding another 2. - Total cost = $2 + 2 = 4$, which matches J_i^3. If Day 1 skips delivery, I drops to 0 (cost 5), and even if Day 2 delivers 2, the total becomes 8, which is suboptimal.

Notes. (i) The third column of A is always $+\infty$ because with $d = 1$ the end-of-day inventory cannot exceed 1. (ii) In general, if evaluating $A(I, J)$ requires enumerating multiple route options $r \in \mathcal{K}$, then $A(I, J) = \min_r A(I, J; r)$, and GPU parallelism naturally extends to three dimensions $(\omega,\ I{\to}J,\ r)$ with reductions first over r and then over I.

5.4 Monte Carlo Method Proposition.

Empirical Risk Minimization (ERM) method is analogous to the Monte Carlo method for estimating a population mean via sample averages and fits naturally within the ERM framework for stochastic programming. Formally, we distinguish between:

- *True problem*:
$$(P) \quad z^* = \min_{x \in \mathbb{X}} \mathbb{E}[f(x, \tilde{\xi})],$$
- *Sample-average problem* with m scenarios:
$$(P_m) \quad z_m^* = \min_{x \in \mathbb{X}} \frac{1}{m} \sum_{i=1}^{m} f(x, \tilde{\xi}^i),$$

As the sample size m increases, the optimal solution x_m^* of the sample problem converges to the true optimal solution x^*, and the optimal value z_m^* approaches z^*. The following result summarizes the fundamental properties of the ERM method under mild regularity conditions (cf. (Shapiro, 2003)).

Proposition 1. *Let $\{\tilde{\xi}^1, \ldots, \tilde{\xi}^m\}$ be i.i.d. samples of $\tilde{\xi}$. Denote by*

$$(P) \quad z^* = \min_{x \in \mathbb{X}} \mathbb{E}[f(x, \tilde{\xi})], \qquad (P_m) \quad z_m^* = \min_{x \in \mathbb{X}} \frac{1}{m} \sum_{i=1}^{m} f(x, \tilde{\xi}^i),$$

the true and sample average problems, respectively, with optimal solutions x^* and x_m^*. Then:

(Bias) $$\mathbb{E}[f(x_m^*, \tilde{\xi})] \leq z^*, \tag{28}$$

(Consistency) $$\mathbb{E}[f(x_m^*, \tilde{\xi})] \xrightarrow{a.s.} z^* \quad \text{as } m \to \infty, \tag{29}$$

(Probabilistic Convergence) $$\lim_{m \to \infty} \Pr\left\{\mathbb{E}[f(x_m^*, \tilde{\xi})] \leq z^* + \tilde{\epsilon}_m\right\} \geq 1 - \alpha, \quad \tilde{\epsilon}_m \downarrow 0, \tag{30}$$

(Rate of Convergence) $$\sqrt{m}\,(z_m^* - z^*) \xrightarrow{d} \mathcal{N}\big(0, \sigma^2(x^*)\big). \tag{31}$$