Batch Learning in Stochastic Dual Dynamic Programming

Daniel Ávila*, Anthony Papavasiliou

Center of Operations Research and Econometrics, Université Catholique de Louvain, 1348 Louvain-La-Neuve, Belgium

Nils Löhndorf

Luxembourg Centre for Logistics and Supply Chain Management, University of Luxembourg, 1511 Luxembourg, Luxembourg

Abstract

We consider the stochastic dual dynamic programming (SDDP) algorithm, which is a widely employed algorithm applied to multistage stochastic programming, and propose a variant using batch learning, a technique used with success in the reinforcement learning framework. We cast SDDP as a type of Q-learning algorithm and describe its application in both risk neutral and risk averse settings. We demonstrate the efficiency of the algorithm on a lost sales inventory control problem with lead times, as well as a real-world instance of the long-term planning problem of inter-connected hydropower plants in Colombia. We find that the proposed technique is able to produce tighter optimality gaps in a shorter amount of time than conventional SDDP, including the PSR SDDP commercial software. We also find that parallel computation of SDDP backward passes benefit from batch learning.

Keywords: Stochastic programming, Dynamic programming, Reinforcement learning, SDDP, Parallel Computing

1. Introduction

Stochastic dual dynamic programming (SDDP) is a scalable algorithm for solving multistage stochastic programming problems. Since the seminal work of Pereira & Pinto (1991), SDDP has captivated the interest of the stochastic programming community and achieved widespread adoption in industrial applications (Flach et al., 2010; De Matos et al., 2010; Pinto et al., 2013; Löhndorf et al., 2013; Löhndorf & Wozabal, 2020; Dowson et al., 2019).

The algorithm selects supporting hyperplanes that provide tight outer approximations of the value functions of the dynamic programming formulation in regions of the state space that can be reached by the optimal policy. While the gap between simulated cost from following the incumbent policy and cost approximation closes quickly for many problems, there exists

*Corresponding author

Email addresses: daniel.avila@uclouvain.be (Daniel Ávila), anthony.papavasiliou@uclouvain.be (Anthony Papavasiliou), nils.loehndorf@uni.lu (Nils Löhndorf)

a number of counter-examples where convergence stalls and the gap does not close, e.g. in long-term hydropower planning (Shapiro et al., 2013).

In order to speed up convergence, several approaches have been proposed in the extant literature. Such methods include cut selection techniques for removing redundant hyperplanes (De Matos et al., 2015; Guigues, 2017; Guigues & Bandarra, 2019; Löhndorf et al., 2013), regularization techniques for selecting better trial points during the forward pass (Asamov & Powell, 2018), and parallel computing schemes (da Silva & Finardi, 2003; Pinto et al., 2013; Helseth & Braaten, 2015; Machado et al., 2021). While each of these approaches individually improves the convergence speed of SDDP, none of them circumvents the fact that the algorithm stalls on certain problems.

In order to overcome this problem, we introduce a novel variant of SDDP, which we refer to as the *Batch Learning SDDP (BL-SDDP)*. The algorithm integrates ideas from reinforcement learning into the SDDP framework and is better suited for parallel computing than conventional SDDP.

In a computational study, we demonstrate that BL-SDDP is able to achieve tighter optimality gaps on problem instances where conventional SDDP fails to close the gap. We further strengthen this result by comparing our implementation with that of the commercial software PSR SDDP which we use to optimize instances of the Colombian long-term hydropower planning problem.

1.1. Batch learning

Reinforcement learning (RL) is an area of machine learning that aims at training computational (robotic) agents to make decisions in a dynamic and uncertain environment so that these agents can maximize their rewards.

Reinforcement learning distinguishes between model-free and model-based methods. In model-free RL, the objective is to train an agent by observing its interaction with an environment for which no model exists. In model-based RL, the objective is to train an agent that interacts with a model of the environment. Since stochastic programming provides us with a model of the environment, we will focus on model-based methods.

In Section 2, we formulate multistage stochastic programming problems as Markov decision processes (MDPs) as they are typically used in the model-based RL literature. We then cast SDDP as a reinforcement learning algorithm that, similarly to Q-learning, evolves by iteratively building approximations of Q-factors.

Conventional RL algorithms apply a sequential strategy that updates a decision policy as soon as new information arrives. It has been found that it can be better to delay and batch these updates so as to avoid costly matrix multiplications when updating gradients or simply to reduce noise (Kalyanakrishnan & Stone, 2007). This so-called batch learning has emerged as an attractive alternative that often outperforms other reinforcement algorithms (Lin, 1992; Kalyanakrishnan & Stone, 2007; Lange et al., 2012).

In Section 3 we introduce experience replay, a batch learning technique. Experience replay (Lin, 1992) resamples states and actions that have been visited in previous iterations, thereby

replacing the old belief regarding the expected cost associated to these state-action pairs with more recent information. In this way, the algorithm decreases the delay to revisit previously explored states which may be of high impact to the problem (Lin, 1992; Lange et al., 2012). Google's DeepMind algorithm uses experience replay in combination with Q-learning as a strategy for obtaining human-level performance in a series of Atari games (Mnih et al., 2015).

We propose to use batch learning and experience replay in order to speed up learning and thereby the convergence of SDDP. The proposed algorithm, which we call *Batch Learning SDDP* (BL-SDDP), applies experience replay during the backward pass, where experiences correspond to the trial points of previous iterations.

In Section 5, we test a series of batching strategies for choosing which experiences to select and study their effectiveness on a number of problem instances where conventional SDDP fails to converge. In particular, we empirically demonstrate that the algorithm is able to achieve tighter optimality gaps than the PSR SDDP commercial implementation. Overall, the results demonstrates encouraging results resulting from the integration of ideas from reinforcement learning with SDDP.

1.2. Parallel computing

Parallel schemes are a natural choice for countering the computational complexity of SDDP (Pinto et al., 2013). In spite of this, the extant literature mostly discusses parallel Monte Carlo sampling during the forward and backward passes of the algorithm (da Silva & Finardi, 2003; Pinto et al., 2013; Helseth & Braaten, 2015; Dowson & Kapelevich, 2017). An obvious downside of increasing the number of parallel forward passes is that it often leads to an accumulation of trial points that are similar, which leads to an accumulation of redundant cuts that hardly improve the approximation but severely slow down convergence of the algorithm (De Matos et al., 2015, 2010).

Batch learning with experience replay offers an attractive alternative for exploiting parallel computing capacities without impeding convergence with redundant cut information. In particular, experience replay reuses trial points from past iterations whose updates can be batched and parallelized during the backwards pass. We describe a parallel scheme for BL-SDDP in Section 4 and compare its efficiency with that of SDDP in Section 5.

2. Problem Formulation

The present section is divided into three subsections. The first subsection provides a brief introduction to Markov Decision Processes (MDP). The second subsection provides a link between multistage stochastic programs, which is the common framework for SDDP, and Markov Decision Processes. Finally, in the last subsection we describe the SDDP algorithm and cast it as an RL algorithm, which is commonly described using MDP notation.

2.1. Markov decision processes

Markov decision processes (MDP) are a framework for analyzing decision making over time and under uncertainty. An MDP is defined by the tuple (S_t, A_t, C_t, P), where S_t is the set

of states, A_t is the set of actions, $C_t : S_t \times A_t \to \mathbb{R}$ is the reward function of an agent, and $P(s_{t+1}|a_t, s_t)$ is the probability of transitioning to state $s_{t+1} \in S_t$ if we are in state $s_t \in S_t$ and select action $a_t \in A(s_t)$. The set $A(s_t)$ indicates the set of feasible actions when in state s_t. A policy $\pi = (\pi_1, \pi_1, \ldots)$ is a vector of functions, where each function maps states to actions[1], namely $\pi_t : S_t \to A_t$. We will focus on finite horizon models with a time horizon T. Further details on the definition of MDPs can be found in Powell (2007); Sutton & Barto (2018); Puterman (2014). The objective in MDPs is to obtain a policy $\pi \in \Pi$ that minimizes the expected cost over the decision-making horizon:

$$\min_{\in} \mathbb{E}\Big[\sum_{t=1}^{T} C_t(s_t, a_t)\Big]$$

The optimal expected reward is often calculated with the help of the value functions of the problem. Concretely, the value function $V_t(s_t)$ is the expected reward when starting in state s_t and following policy π. Mathematically, value functions are defined as

$$V_t(s_t) = \mathbb{E}\Big[\sum_{n=t}^{T} C_n(s_n, a_n)\Big| s_t\Big]$$

We will use the notation,

$$V_{t+1}(s_t, a_t) = \mathbb{E}\Big[V_{t+1}(s_{t+1})|s_t, a_t\Big]$$

to refer to the value functions after taking expectation. The value functions that correspond to an optimal policy satisfy the following optimality conditions (Sutton & Barto, 2018; Puterman, 2014):

$$\begin{aligned} V_t^*(s_t) &= \min_{a_t \in \mathcal{A}(s_t)} C_t(s_t, a_t) + \mathbb{E}\Big[V_{t+1}^*(s_{t+1})|s_t, a_t\Big] \\ &= \min_{a_t \in \mathcal{A}_t(s_t)} C_t(s_t, a_t) + V_{t+1}^*(s_t, a_t) \end{aligned}$$

As we describe subsequently, the SDDP algorithm proceeds by approximating $V_{t+1}^*(s_t, a_t)$ using supporting hyperplanes and then using the optimality equation in order to approximate the value functions of preceding stages.

Another noteworthy concept within the reinforcement learning framework are Q-factors. A Q-factor $Q(s_t, a_t)$ represents the expected reward that is obtained after selecting action a_t while being in state s_t, and then following policy π. Formally, we can define Q-factors as

$$Q_t(s_t, a_t) = \mathbb{E}\Big[\sum_{n=t}^{T} C_n(s_n, a_n)\Big| s_t, a_t\Big].$$

[1]The literature presents more general policies, where a state is mapped to a probability measure over the set of actions. However we will focus on deterministic policies in this paper.

Note that we can express our Q-factors in terms of the value functions as

$$\begin{aligned} Q_t(s_t, a_t) &= C_t(s_t, a_t) + \mathbb{E}\left[V_{t+1}(s_{t+1}) | s_t, a_t\right] \\ &= C_t(s_t, a_t) + V_{t+1}(s_t, a_t) \end{aligned} \quad (1)$$

Note also that the optimal value functions can be expressed in terms of the optimal Q-factors as

$$V_t^*(s_t) = \min_{a_t \in \mathcal{A}(s_t)} Q_t^*(s_t, a_t) \quad (2)$$

2.2. Multistage stochastic programming and MDP

Multistage stochastic linear programs and the algorithms used to tackle them such as SDDP, are commonly described using stochastic programming terminology. We indicate the connections between multistage stochastic linear programs and MDPs using the familiar (to the SDDP community) example of a hydrothermal scheduling problem which unfolds in two stages.

Example 2.1. *Consider a two-stage hydrothermal scheduling problem given by:*

$$\begin{aligned} \min\ & C \quad g_1 + VOLL \quad ls_1 + \mathbb{E}[C \quad g_2(\xi_2) + VOLL \quad ls_2(\xi_2)] \\ s.t.\ & q_1 + g_1 + ls_1 = L_1 \\ & x_1 = X_0 + A_1 \quad q_1 \\ & q_2(\xi_2) + g_2(\xi_2) + ls_2(\xi_2) = L_2 \\ & x_2(\xi_2) = x_1 + A_2(\xi_2) \quad q_2(\xi_2) \\ & g_t(\xi_t) \quad \bar{G} \\ & x_t(\xi_t) \quad \bar{X} \\ & g_t(\xi_t), x_t(\xi_t), q_t(\xi_t) \quad 0, \ \textit{for all } \xi_t \in \Omega_t \end{aligned}$$

Here, g_t is the power generated by thermal units at a marginal cost C. The thermal generators can produce up to $\bar{G}$ units of power per period. The system can shed load at a high cost $VOLL$, and ls_t is the amount of power that is curtailed from consumers. The variable q_t is the power generated from hydro units, generated at zero cost. The variable x_t represents the amount of available energy in the hydro reservoir at the end of period t. We assume an initial condition X_0 for the reservoir. The hydro reservoir can store a maximum amount $\bar{X}$ of energy. The system is subject to uncertain rainfall represented by A_t. We assume there are finitely many outcomes Ω_t. Note that there is a single realization $\Omega_1 = \{\xi_1\}$ in the first stage. Using standard MDP notation, we can write the problem as follows.

- *The set of states is defined as*

$$
\begin{aligned}
S_1 &= \{(X_0, \xi_1)\} \\
S_2 &= \Big\{(x_1, \xi_2) : \textit{ there exists } q_1, g_1 \textit{ s.t. } x_1 = X_0 + A_1 \quad q_1 \\
&\qquad q_1 + g_1 + ls_1 = L_1 \\
&\qquad g_1 \quad \bar{G}, x_1 \quad \bar{X}, g_1, x_1, q_1 \quad 0, \xi_2 \in \Omega_2\Big\}
\end{aligned}
$$

- *The set of actions for a state $s_t = (x_{t-1}, \xi_t)$ is defined as*

$$
\begin{aligned}
\mathcal{A}_t(s_t) = \{(x_t, q_t, g_t, l_t) : \; & q_t + g_t + ls_t = L_t \\
& x_t = x_{t-1} + A_t(\xi_t) \quad q_t \\
& g_t \quad \bar{G}, x_t \quad \bar{X}, g_t, x_t, q_t \quad 0\}
\end{aligned}
$$

  We will use a_t to refer to an action.

- *The reward function is given by*

$$C_t(s_t, a_t) = C \quad g_1 + VOLL \quad ls_t$$

- *We can define the probability of transitioning to state s_2 while being in state $s_1 = (X_0, \xi_1)$ and selecting action $a_t = (x_t, q_t, g_t, l_t)$ as follows.*

$$
P(s_{t+1} \mid s_t, a_t) = \begin{cases} P(\xi_2 \mid \xi_1) & s_2 = (x_t, \xi_{t+1}) \\ 0 & \textit{otherwise} \end{cases}
$$

Furthermore, the dynamic programming equations can be described as follows,

$$
\begin{aligned}
V_t^*(s_t) &= \min_{a_t \in \mathcal{A}(s_t)} C \quad g_t + VOLL \quad ls_t + \mathbb{E}\Big[V_{t+1}^*(s_{t+1}) \mid s_t, a_t\Big] \\
&= \min_{a_t \in \mathcal{A}(s_t)} C \quad g_t + VOLL \quad ls_t + \mathcal{V}_{t+1}^*(s_t, a_t)
\end{aligned}
$$

The Q-factors satisfy,

$$Q_t^*(s_t, a_t) = C \quad g_t + VOLL \quad ls_t + \mathcal{V}_{t+1}^*(s_t, a_t)$$

Now let us consider the general case of a multistage stochastic linear program over T stages. For each stage we have vectors u_t, v_t, b_t as well as matrices B_t, A_t, D_t that will form the stochastic data process $\xi_t = (u_t, v_t, b_t, B_t, A_t, D_t)$. We assume that u_1, v_1, b_1, A_1, C_1 are deterministic. Let us assume that, at each stage, there are finitely many outcomes Ω_t, and that the data process follows a Markov chain. We therefore consider that we can define transition matrices $P(\xi_{t+1} \mid \xi_t)$.

For each stage t and uncertainty realization $\xi_t \in \Omega_t$ we define the feasibility set

$$\text{Feas}_t(x_{t-1}, \xi_t) = \{(x_t, y_t) \in \mathbb{R}^n : x_t, y_t \geq 0, B_t(\xi_t)x_{t-1} + A_t(\xi_t)x_t + D_t(\xi_t)y_t = b_t(\xi_t)\}$$

The considered MDP is given by the tuple (S_t, X_t, C_t, P).

- The set of states is defined as

$$S_1 = \{(x_0, \xi_1)\}$$

$$S_t = \Big\{(x_{t-1}, \xi_t) : \xi_t \in \Omega_t \text{ and } (x_{t-1}, y_{t-1}) \in \text{Feas}_{t-1}(x_{t-2}, \xi_{t-1}) \text{ for some } (x_{t-2}, \xi_{t-1}) \in S_{t-1}\Big\}$$

  We can restrict ourselves to $(x_{t-1}, y_{t-1}) \in \text{Feas}_{t-1}(x_{t-2}, \xi_{t-1})$ that are vertices of the feasibility set. Therefore, since Ω_t is finite, we have finite states.

- The set of actions for a state $s_t = (x_{t-1}, \xi_t)$ is defined as

$$A_t(s_t) = \text{Feas}_t(x_{t-1}, \xi_t).$$

  To improve readability, we define $a_t = (x_t, y_t)$ to refer to an action. As before, we restrict ourselves to actions that correspond to a vertex of the feasibility set. Thus, there are finitely many actions to consider. The definition using finite states and actions is for ease of exposition and in order to avoid defining a probability measure over a continuous space.

- The reward function is given by

$$C_t(s_t, a_t) = u_t(\xi_t)^T x_t + v_t(\xi_t)^T y_t$$

- The function defining the dynamics is expressed as $f_t(s_t, a_t, \xi_{t+1}) = (x_t, \xi_{t+1})$. Furthermore, we can define the probability of transitioning to state s_{t+1} while being in state $s_t = (x_{t-1}, \xi_t)$ and selecting action $a_t = (x_t, y_t)$ as follows.

$$P(s_{t+1} | s_t, a_t) = \begin{cases} P(\xi_{t+1} | \xi_t) & s_{t+1} = (x_t, \xi_{t+1}) \\ 0 & \text{otherwise} \end{cases}$$

With these definitions, we can pose the MDP problem as one of finding a policy that minimizes the expected reward, that is to say,

$$\min_{\pi \in \Pi} \mathbb{E}\Big[\sum_{t=1}^{T} C_t(s_t^{\pi}, a_t^{\pi})\Big] \tag{MDP-P}$$

Furthermore, the problem can be written using stochastic programming notation as

$$\min_{\substack{A_1x_1+D_1y_1=b_1\\ x_1,y_1\geq 0}} u_1^T x_1 + v_1^T y_1 + \mathsf{E}\left[\min_{\substack{B_2x_1+A_2x_2+D_2y_2=b_2\\ x_2\geq 0}} u_2^T x_2 + v_2^T y_2 + \mathsf{E}\left[\quad + \mathsf{E}\left[\min_{\substack{B_Tx_{T-1}+A_Tx_T+D_Ty_T=b_T\\ x_T\geq 0}} u_T^T x_T + v_T^T y_T\right]\right]\right] \tag{MSP-P}$$

To link reinforcement learning with stochastic dual dynamic programming, we establish the following relationship.

Lemma 2.2. *The problems MDP-P and MSP-P are equivalent.*

The proof of the lemma is available in the appendix.

2.3. Stochastic dual dynamic programming as a reinforcement learning algorithm

As discussed in Shapiro et al. (2013), the optimal expected value functions $V^*_{t+1}(s_t, a_t)$ (which correspond to the expected cost-to-go functions in stochastic programming terminology) can be outer-approximated by a piece-wise linear function approximation $V_{t+1}(s_t, a_t)$. Such an approximation is obtained through supporting hyperplanes H, commonly referred to as cuts. These cuts are computed by calculating the sub-gradient of the expected value function. The idea of SDDP is to iteratively generate an approximation of the expected value function. Each iteration of the SDDP algorithm is a two-step process that consists of forward and backward passes. During forward passes, a sample of uncertainty is chosen. As the algorithm evolves forward in the number of stages, trial actions are obtained. Backward passes evolve backwards in the number of stages. During backward passes, we generate cuts for the expected value functions at the trial actions that are obtained in the forward pass (Shapiro et al., 2013). Concretely, we can describe SDDP as a double-pass algorithm. This is a class of algorithms that are found in the MDP literature (Powell, 2007).

Lemma 2.3. *A model-based double-pass algorithm is equivalent to SDDP with the proper update rule.*

Proof. Due to equation (1), we know that the Q-factors satisfy

$$Q^*_t(s_t, a_t) = C_t(s_t, a_t) + V^*_{t+1}(s_t, a_t)$$

Moreover, V^*_{t+1} can be approximated by building a supporting hyperplane around x^n_t (Shapiro et al., 2013), where $(x^n_t, y^n_t) = a^n_t$ is a trial action. Thus, we can consider an update rule that adds a supporting hyperplane around x^n_t to V_{t+1} and updates the Q-factor as $Q_t(s_t, a_t) = C_t(s_t, a_t) + V_{t+1}(s_t, a_t)$. Building such a supporting hyperplane requires a model-based scheme. A detailed exposition on how the supporting hyperplane is built can be found in the appendix. Let us express this update rule as $Q_t = U(a_t, Q_{t+1})$. We can now formulate the SDDP algorithm as a double-pass algorithm:

1. Provide an initialization of the Q-factors $Q_t^0(s_t, a_t)$ for $s_t \in S_t$, $a_t \in A_t(s_t)$ and $t = 1, \ldots, T$.
2. For $n = 1, \ldots, N$
   (2.1) **Forward Pass**: Initialize at a state s_1.
      (2.1.1) For $t = 1, \ldots, T$
         (2.1.1.1) Find the decision using the current Q-factors.

$$a_t^n \in \arg\min_{a_t \in \mathcal{X}(s_t^n)} Q_t^{n-1}(s_t^n, a_t)$$

         (2.1.1.2) Take action a_t^n and transition to state s_{t+1}^n.
   (2.2) **Backward Pass**:
      (2.2.1) For $t = T, \ldots, 1$
         (2.2.1.1) Update Q_t^{n-1} using the update rule.

$$Q_t^n = U_t(a_t^n, Q_{t+1}^n)$$

3. Return the Q_t^N estimates for $t = 1, \ldots, T$

□

Remark 2.4. *Double-pass algorithms, with the proper update rule, include commonly known reinforcement learning algorithms such as TD(1) (Powell, 2007). Furthermore, single-pass algorithms, where just a forward pass is applied and the updates evolve forward in time, include algorithms such as Q-learning or more generally TD(0) algorithms (Powell, 2007).*

An attractive way to graphically understand the SDDP algorithm is by relying on a lattice representation (see definition 1), which is a graphical way of representing the underlying structure of the problem. The SDDP algorithm over a lattice is described in Figure 1. The Figure presents a lattice with 3 stages. The first stage corresponds to a singe realization and the other stages correspond to 3 realizations.

Definition 1. *A lattice is an undirected graph. Each column of nodes represents a stage of the problem and each node represents an uncertainty realization. If a node has an uncertainty realization ξ_t associated to it, then the node also has an optimization problem associated to it. This problem corresponds to the calculation of the value function $V_t(x_{t-1}, \xi_t)$, obtained through equation (2). Note also that each node has an expected value function associated to it.*

Figure 1 presents the SDDP algorithm as it traverses the lattice. The computational time evolves along the x-axis. The forward pass starts by sampling a scenario. The transition probabilities are used for this purpose, selecting as a result the nodes that are indicated with a red dashed box, namely nodes 1,4,7. The value function V_t corresponding to these nodes is calculated, starting from the first, and ending with the last stage, that is to say beginning with node 1, then 4, and finally 7. As a result, we obtain trial points $\widehat{x}_1, \widehat{x}_2, \widehat{x}_3$. The backward pass

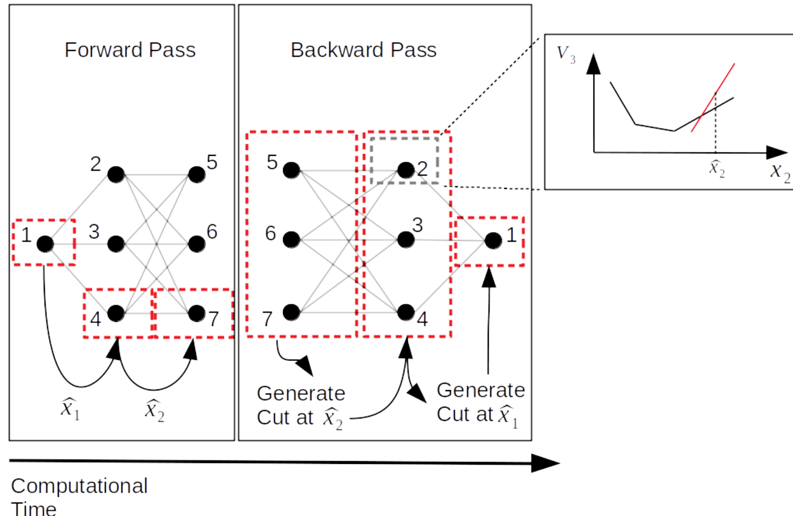


Figure 1: The SDDP algorithm described graphically over a lattice. The computational time evolves along the x-axis. The length of the red dashed boxes represents the elapsed computational time. In the forward pass, a scenario is drawn and trial points are obtained. The backward pass starts by solving the last stage, which produces a cut for the expected value functions corresponding to the nodes of stage 2. The backward pass continues in this manner throughout the remaining stages.

is then executed. Starting from the last stage, the value functions for all nodes, evaluated at the trial point $\widehat{x}_2$, are calculated. In Figure 1, this corresponds to solving nodes 5,6 and 7. The solution information of the nodes is used to build a cut of the expected value function at the trial point $\widehat{x}_2$. The backward pass then proceeds in the same way for stage 2.

3. Batch Learning SDDP (BL-SDDP)

The experience replay scheme has gained popularity in the reinforcement learning framework due to its success when employed with other reinforcement learning algorithms (Lin, 1992; Pieters & Wiering, 2016; Mnih et al., 2015). This section presents, as a corollary of the results presented in section 2, a novel application of the experience replay framework in the context of SDDP. The first subsection presents the experience replay scheme, while the second subsection introduces our novel batch learning SDDP algorithm.

3.1. Experience replay

Classical methods in reinforcement learning update Q_t^{n-1} based on the currently computed states and actions, commonly referred to as experiences, namely s_t^n, a_t^n, s_{t+1}^n. After the update, these values are discarded and the algorithm proceeds with the next iteration. The experience replay framework, introduced first by Lin (1992), aims to update Q_t using previously computed experiences. The motivation behind this is that updating a state-action pair (s_t, a_t) at stage t may affect some preceding states s_{t-1}. Nevertheless, this will not back propagate until state s_{t-1} is re-visited. Furthermore, states preceding s_{t-1} will need to be re-visited after the update of s_{t-1} before being able to see the update carried out in the upper layers. Therefore, the

back-propagation of information is not possible unless states are re-visited. Given an arbitrary update rule $Q_t = U_t(s_t, a_t, s_{t+1}, Q_{t+1})$, the experience replay algorithm, which has been adapted for the finite horizon setting, can be described as follows (Lin, 1992; Pieters & Wiering, 2016; Mnih et al., 2015).

1. Provide an initialization of the Q-factors $Q_t^0(s_t, a_t)$ for $s_t \in S_t$, $a_t \in A_t(s_t)$ and $t = 1, \ldots, T$. For each stage, provide a set of experiences $M_t = \{(s_t^n, a_t^n, s_{t+1}^n) : n = 1, \ldots, N\}$ and let K be the batch size $K \leq |M_t|$, namely the number of experiences to replay.
2. For $t = T, \ldots, 1$
   (2.2) Retrieve a subset $\{(s_t^k, a_t^k, s_{t+1}^k) : k = 1, \ldots, K\} \subseteq M_t$. For $k = 1, \ldots, K$
      (2.2.1) Update Q_t^0 using the update rule.

$$Q_t^1 = U_t(s_t^k, a_t^k, s_{t+1}^k, Q_{t+1}^1)$$

3. Return the Q_t^1 estimates for $t = 1, \ldots, T$

Note that this process can be repeated iteratevely. That is to say, a set of experiences are collected, the experience replay algorithm is applied, afterwards more experiences are collected, and the experience replay algorithm is applied again. In the literature, when K equals the total set of experiences, the method is usually referred to as a full-batch update, whereas when K is less than the total size it is referred to as a mini-batch update.

3.2. BL-SDDP description

As presented in section 2.3, the SDDP algorithm can be described as a type of double-pass algorithm, of the sort that can be encountered in the reinforcement learning literature (lemma 2.3). As a consequence, we can apply known techniques for MDP algorithms, such as the experience replay scheme. This leads to the Batch Learning SDDP algorithm described as follows.

1. Provide an initialization of the Q-factors $Q_t^0(s_t, a_t)$ for $s_t \in S_t$, $a_t \in A_t(s_t)$ and $t = 1, \ldots, T$. Let K be the batch size and let Z be the number of collected experiences before applying experience replay. Let $M = \emptyset$ be the set of experiences.
2. For $n = 1, \ldots, N$
   (2.1) Apply SDDP, collecting up to Z experiences, and add them to the replay memory M.
   (2.2) Consider a batch of K experiences of M. Apply experience replay on the K experiences.
3. Return the Q_t approximations.

Note that, as we are using the update rule based on supporting hyperplanes, the replay memory simply requires $M = \{a_t^n : n = 1, \ldots, N\}$. The proposed scheme can also be seen as an update of cuts, which is carried out for a batch of K cuts every Z SDDP iterations. Relative to standard SDDP, the improved performance of BL-SDDP stems from the fact that it approximates the

value functions more diligently in the backward pass. It thus avoids the back-propagation of approximation errors of value functions at later stages of the problem to “contaminate” the approximations of value functions at earlier stages of the algorithm. This comes at the cost of increased computational effort at each backward pass relative to standard SDDP. We propose resorting to parallel computing in order to cope with this increased computational burden, and thus to combine the most appealing attributes of experience replay and SDDP into a single and highly parallelizable algorithmic procedure.

4. SDDP Parallelization Strategies

4.1. Standard SDDP parallelization

The parallel SDDP literature presents parallelization schemes that are mostly focused on increasing the number of Monte Carlo forward samples and distributing these samples among the available processors (da Silva & Finardi, 2003; Pinto et al., 2013; Helseth & Braaten, 2015; Dowson & Kapelevich, 2017). This common SDDP parallelization framework can be described as follows:

- Forward pass: The forward pass consists of N Monte Carlo scenarios. Each processor computes a different scenario, thereby producing trial points x_1^n, , x_T^n, for $n = 1$, , N.
- Backward pass: At stage t, the n-th processor generates a cut for the expected value functions of stage t 1 at point x_{t-1}^n.

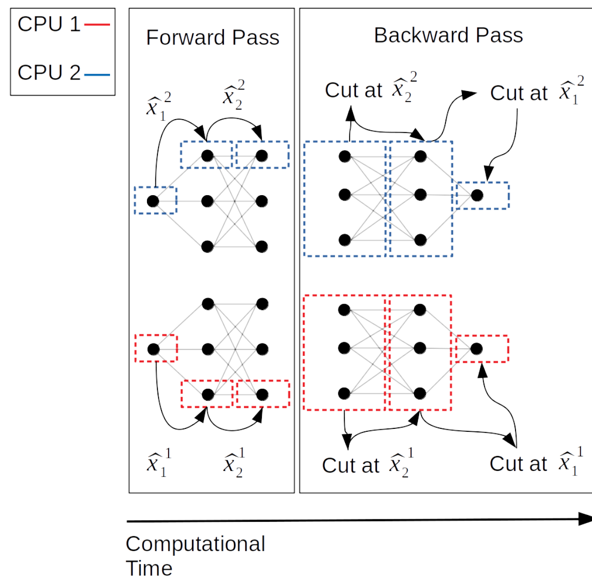


Figure 2: Standard Parallel SDDP Scheme. The forward pass consists of several sample paths, which are distributed among the available processors. The backward pass distributes the trial points that are obtained in the forward pass among the processors.

In Figure 2 we depict this procedure graphically over a lattice. The x-axis represents the elapsed computing time. In the forward pass the red CPU draws a scenario implied by the

lattice distribution. The red CPU then proceeds to solve the node problems and thus produces trial points. The red CPU then proceeds to the backward pass. During the pass the cuts are built around the trial points found by the red CPU. The blue CPU follows a similar sequence of steps.

The communication between processors in this scheme has commonly been synchronous, and is discussed in a number of publications (Pereira & Pinto, 1991; da Silva & Finardi, 2003; Pinto et al., 2013; Helseth & Braaten, 2015; Machado et al., 2021). Synchronization is included at each stage of the backward pass. Thus, at the end of each stage the cuts are shared among the processors.

The literature has proposed certain variants in the communication between processors. In Helseth & Braaten (2015) the authors present a relaxation of the synchronization points, whereby a processor waits for only a subset of processors before proceeding with the next stage of the backward pass. The authors describe the benefits of their proposal relative to a fully synchronous version. In Dowson & Kapelevich (2017) the authors propose an asynchronous version, whereby a processor does not wait for other processors to compute a cut before proceeding to the next stage in the backward pass. Nevertheless, the advantages of such an approach are not clearly addressed. As past research has demonstrated, that different choices of Monte Carlo sampling may produce different results in the performance of the algorithm (De Matos et al., 2015, 2010). Moreover, as observed by Machado et al. (2021), an increase in Monte Carlo samples has the disadvantage of increasing the workload of the problem. At each iteration the algorithm requires computing more scenarios, thereby resulting in instances where more CPUs imply a greater workload.

The parallelization strategy described in Machado et al. (2021) differs notably from the aforementioned parallel schemes, which are based on increasing the Monte Carlo samples in order to exploit parallelism. In Machado et al. (2021), the authors propose a scheme whereby a processor is attached to a stage and builds cuts for the given stage. Thus, there is no actual backward pass. Instead, all the stages of the backward pass are calculated simultaneously. The authors report advantages relative to the standard parallel SDDP scheme for certain instances. Other instances present a similar behaviour to the standard SDDP algorithm.

As the parallel scheme described in Figure 2 is the most commonly adopted strategy for parallelizing SDDP, we use it as a benchmark for the parallel strategy that we develop in the next subsection.

4.2. Parallelization of BL-SDDP

We propose a novel parallelization scheme for SDDP based on the BL-SDDP algorithm that we propose in section 3. The developed BL-SDDP algorithm is divided into two parts. The first part is a common SDDP run. The second part employs the experience replay framework. We propose a parallelization strategy for each one of these parts.

The SDDP parallelization proceeds as follows. As we have discussed, the standard parallelization strategy for SDDP relies on an increase in the number of Monte Carlo samples in the forward pass. As indicated previously, this is not necessarily beneficial. In order to avoid a

possible deterioration in performance, our proposed parallelization proceeds by fixing a small number of samples in the forward pass. During the forward pass, each sample is distributed to a different processor. During the backward pass, at each stage, the node problems are distributed among the available processors, which means that the different trial points are computed by different processors.

We present an example of this procedure in Figure 3. The forward pass consists of 2 samples. Each processor then computes a sample. Note that the blue CPU remains idle at this point. During the backward pass, at every stage, there are 6 problems, since there are 2 samples and 3 nodes per stage in the lattice. These 6 problems are then distributed among the available processors. Once the problems of the stage have been computed, the processors synchronize, the cuts are built, and the obtained cuts are shared among the available processors.

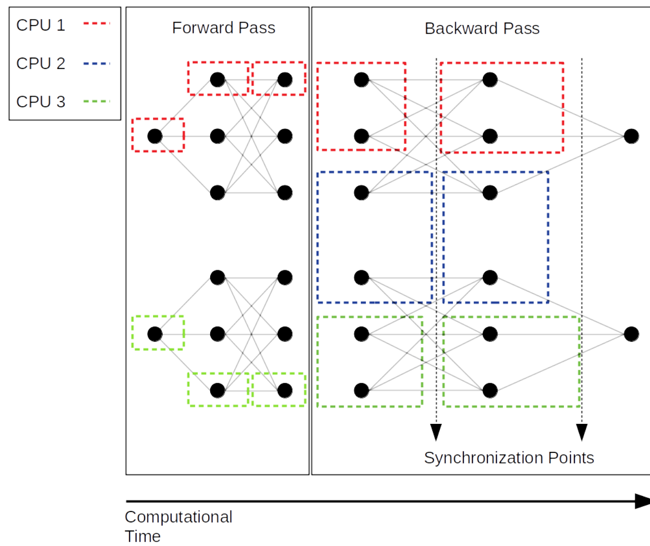


Figure 3: Parallelization of the BL-SDDP algorithm. The SDDP iterations are parallelized by considering a small number of samples in the forward pass, which are then distributed among the available processors. In the backward pass, the problems at each stage are distributed among the processors.

Having described the parallelization procedure for the SDDP iterations in the BL-SDDP framework, we can now describe the parallelization of experience replay. Let us assume that the replay memory is given by $M_t = fa_t^n : n = 1, \ldots, Ng$ where $a_t^n = (x_t^n, y_t^n)$ is an action. Let us further assume a batch of size K. We can easily parallelize the experience replay scheme by distributing the batch of K experiences among the available processors. Then, proceeding backwards in time, each processor updates the Q-factors around the corresponding experiences. That is to say, each processor builds supporting hyperplanes for the expected value functions around the experiences that it receives. Note that, at the end of each stage, the processors synchronize in order to receive the cuts obtained by other processors. This procedure naturally scales up with the introduction of additional CPUs, since additional processors imply that each processor will have a smaller set of experiences.

5. Case Studies

We carry out computational experiments over an inventory control problem with lead times and lost sales, as well as a realistic instance of a long-term planning problem for a network of hydropower plants in Colombia. Our experimental results can be summarized as follows.

i The BL-SDDP algorithm is able to produce tighter gaps in less time, compared to the standard SDDP algorithm and the PSR SDDP commercial implementation.

ii The BL-SDDP algorithm is applicable for both risk neutral and risk averse formulations of multistage stochastic programming. In both cases, it exhibits superior performance relative to the SDDP algorithm.

iii The BL-SDDP parallel scheme is amenable to parallel computing. It exhibits superior performance compared to standard parallel SDDP.

We proceed by briefly introducing the test cases that we analyze in our paper. We provide additional details regarding the models in the appendix.

Inventory Control Problem: The inventory control problem aims at maximizing the expected profits of an inventory manager. Demand in this supply chain is satisfied from on-hand inventory $v_{t-1,n}$, over $t \in 1 \ldots T$ stages and $n \in 1 \ldots N$ products, by selling a quantity s_{tn} of each product at a certain price P. The ordered quantity of product n at stage t is given by x_{tn}. The problem is modeled with lead times. Lead times imply that a certain order quantity x_{tn} is not available until l periods transpire. The uncertainty of the model stems from the demand, which is modeled as a Markov chain. The state space dimension is given by the number of products plus an additional dimension for each product and each delay. Thus, the state space dimension is given by $N \cdot l + N$. The considered model leads to a multistage stochastic program. We consider an instance of the problem with 100 nodes per stage and a total of 50 stages. We consider a setting with 1 product and a 5-period lead time for the order quantities. We thus arrive to a formulation with a 6-dimensional state space.

Hydrothermal Scheduling Problem: The SDDP literature has focused extensively on hydrothermal scheduling problems due to their practical relevance (Pereira & Pinto, 1991; Shapiro et al., 2013; Pinto et al., 2013; De Matos et al., 2015; Löhndorf & Shapiro, 2019). The objective of the problem is to determine optimal storage levels for the hydro reservoirs of a power system under rainfall inflow uncertainty, while respecting operational constraints and satisfying demand, in such a way that the total expected operational costs of the system are minimized. The considered test case is an instance of the Colombian power system that has been made available to the authors by PSR. The case study comprises a network of 32 thermal plants and 42 hydro plants, 25 of which have storage capacity. The case study considers a river network that consists of a 54-dimensional inflow vector. This leads to a high-dimensional multi-stage stochastic program.

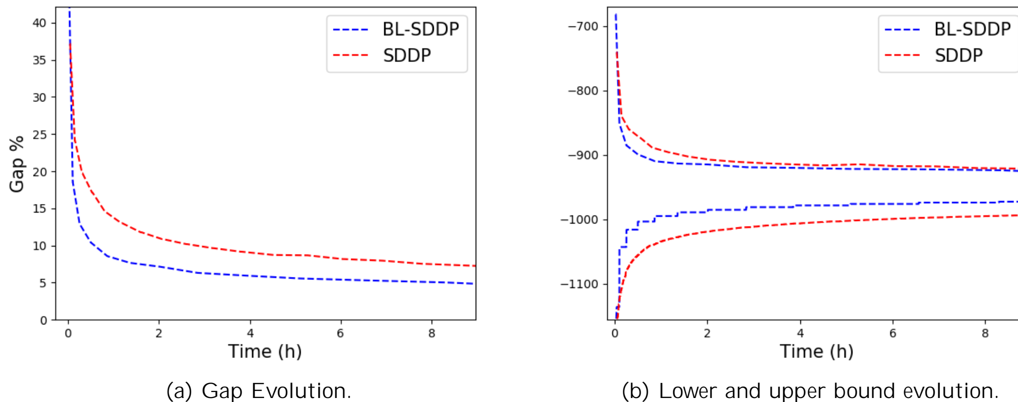
Our algorithms are implemented in Julia v0.6 (Bezanson et al., 2017) and JuMP v0.18 (Dunning et al., 2017). The chosen linear programming solver is Gurobi 8. In order to avoid confusion in the subsequent sections regarding which codes are being compared, the following nomenclature is adopted:

- PSR SDDP: This scheme refers to the PSR software. The language used to code PSR SDDP is FORTRAN and uses the XPRESS solver.
- SDDP: Refers to our base Julia SDDP implementation.
- BL-SDDP: Refers to our proposed algorithm, which we describe in section 3.

5.1. Comparison of BL-SDDP to SDDP

The present subsection aims at analyzing the BL-SDDP algorithm by evaluating it against the standard SDDP scheme. For this purpose, we resort to the same base Julia implementation. The computational work is performed on the Lemaitre3 cluster of UCLouvain, which is hosted at the Consortium des Equipements de Calcul Intensif (CECI). The cluster, where the algorithms are run, consists of 80 compute nodes with two 12-core Intel SkyLake 5118 processors at 2.3 GHz and 95 GB of RAM (3970MB/core), interconnected with an OmniPath network (OPA-56Gbps).

Figures 4 and 5 present the convergence evolution for both test cases when using 20 CPUs. The former figure corresponds to the inventory test case, while the latter corresponds to the hydrothermal problem. The algorithms are set up to compute, at each iteration, 20 samples in the forward pass. The BL-SDDP algorithm performs full-batch updates every 10 iterations for the inventory test case and every 5 iterations for the hydrothermal test case. The figures demonstrate that the BL-SDDP algorithm produces considerably tighter gaps throughout the execution of the algorithm.

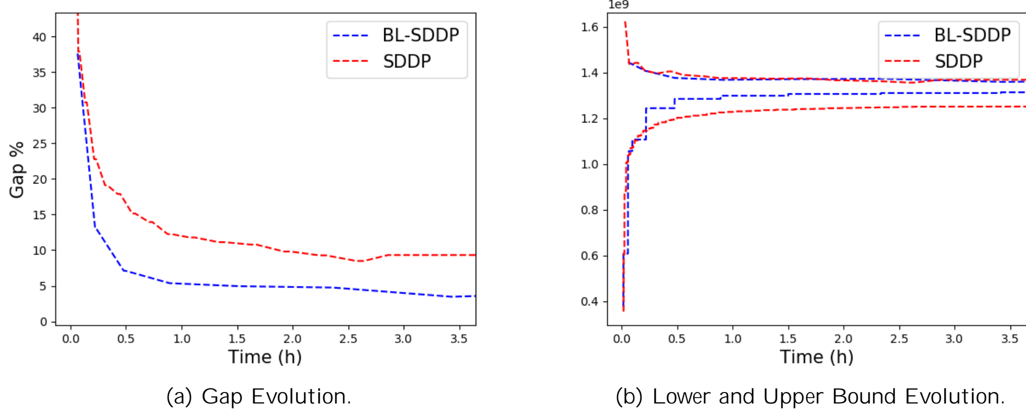


(a) Gap Evolution.

(b) Lower and upper bound evolution.

Figure 4: Convergence evolution for the inventory test case using 20 CPUs.

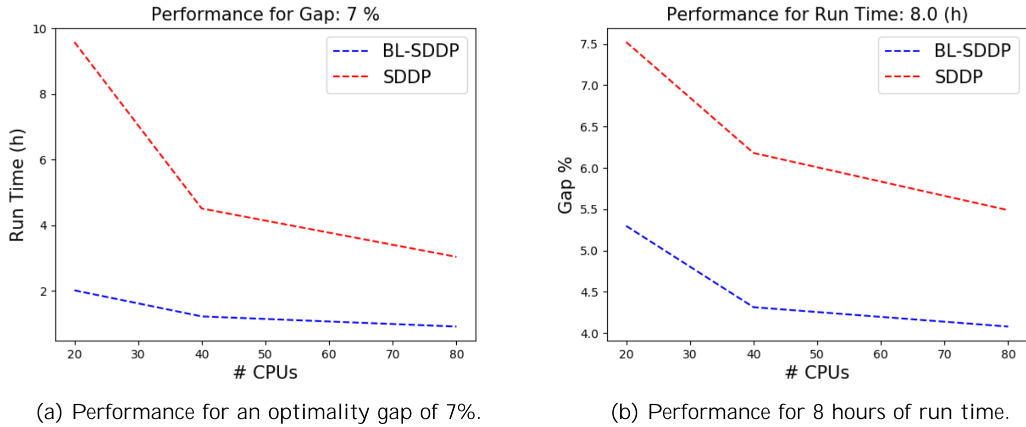
Increasing the number of CPUs produces a similar behaviour. Figure 6 presents the scaling of the algorithm with respect to an increasing number of CPUs for the inventory test case, while Figure 7 presents the corresponding results for the hydrothermal test case. Panel (a) of the



(a) Gap Evolution. (b) Lower and Upper Bound Evolution.

Figure 5: Convergence evolution for the hydrothermal test case using 20 CPUs

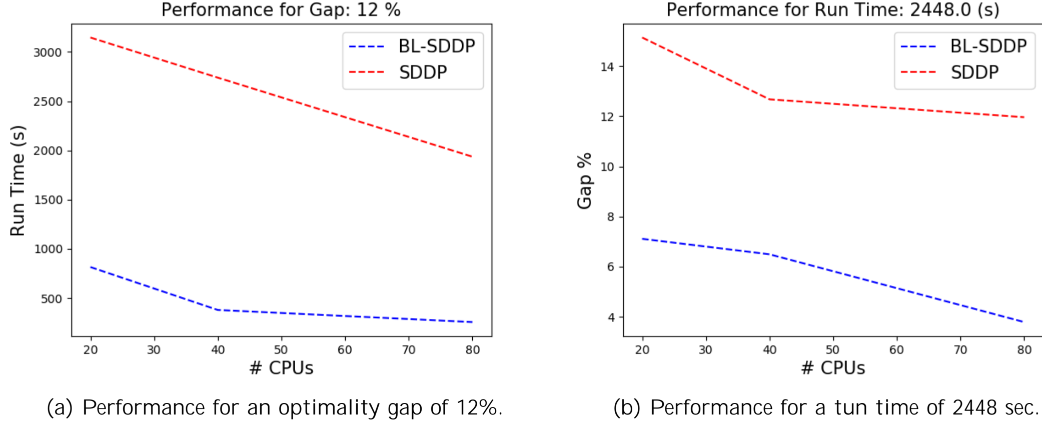
figures presents the elapsed time until achieving a certain target optimality gap in the y-axis. Panel (b) presents the obtained gap after a fixed run time in the y-axis. As we can observe, the BL-SDDP scheme is able to attain a considerable improvement relative the standard SDDP scheme.



(a) Performance for an optimality gap of 7%. (b) Performance for 8 hours of run time.

Figure 6: Effect of CPU increase for the inventory test case. The x-axis corresponds to the number of CPUs. Panel (a) presents the run time for attaining a fixed optimality gap. Panel (b) presents the obtained gap after a fixed run time.

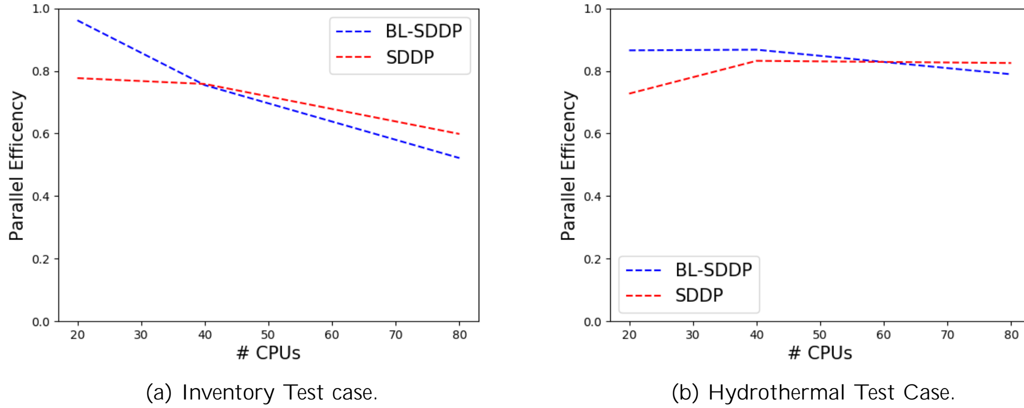
A common measure for estimating the scalability of a parallel algorithm is the so-called Parallel Efficiency (PE), which is defined as $PE = S_T/(N \quad P_T)$. Here, P_T is the parallel time using N cores and S_T is the best serial time. Note that the best possible outcome would be a parallel efficiency of 1. Figure 8 presents the parallel efficiency results for both test cases. We can observe that both SDDP and BL-SDDP achieve a parallel efficiency near 0.8 for the hydrothermal test case. For the inventory test case, we observe a parallel efficiency near 0.8 when using up to 40 CPUs. Once we reach 80 CPUs, the efficiency decreases considerable for both SDDP and BL-SDDP, and is close to 0.6.



(a) Performance for an optimality gap of 12%.

(b) Performance for a tun time of 2448 sec.

Figure 7: CPU increase for the hydrothermal test case.



(a) Inventory Test case.

(b) Hydrothermal Test Case.

Figure 8: Parallel efficiency. Panel (a) shows the results for the inventory problem while panel (b) shows the results for the hydrothermal problem.

5.2. Batch choices

The present subsection aims at providing a brief study on the effect of different batch choices. We will consider the following rules for selecting a batch.

F Corresponds to a full batch update.

R Corresponds to a random batch.

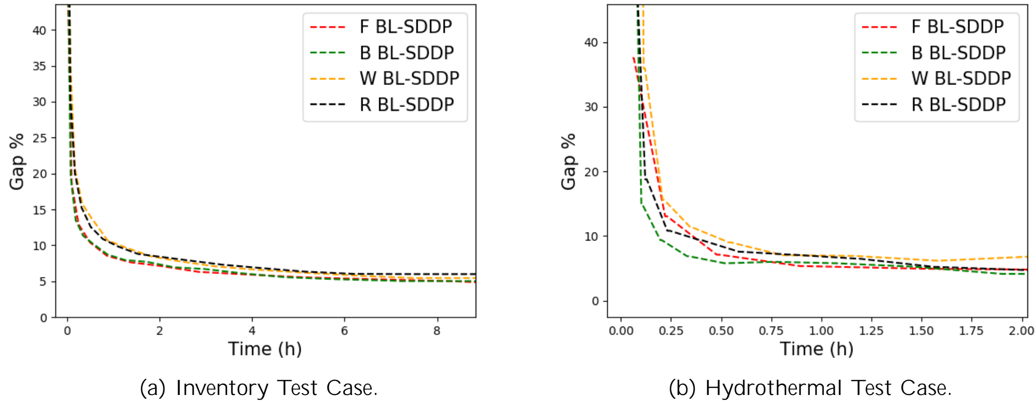
B Let C_i correspond to a cut calculated around x_i. Let δ_i be defined as the distance between the cut C_i and the value function at point x_i, namely:

$$\delta_i = V(x_i) \quad C_i(x_i)$$

The batch corresponds to the smallest δ_i, that is to say, the best cuts.

W Using the same notation as in previous item, the batch corresponds to the highest δ_i. These are in effect the worst cuts.

The results are presented in Figure 9. As in the previous experiments, 20 samples are considered in the forward pass. The batch update is performed every 10 iterations for the inventory test case, and every 5 iterations for the hydrothermal test case. The chosen batch size for batch choices R, B and W is 50%.



(a) Inventory Test Case.

(b) Hydrothermal Test Case.

Figure 9: Batch choices for BL-SDDP. The batch size is set up to 50%, except for the F BL-SDDP that performs a full-batch update.

Figure 9 shows that the W and R strategies tend to behave the worst. On the inventory test case we notice no remarkable difference between the F and B strategies. The hydrothermal test case, on the other hand, presents a significant improvement on the B strategy at the early steps, as compared to the F strategy. Nevertheless, in the long run time both strategies behave similarly. We highlight that several strategies for batch selection may be available. This will be a focus for future research.

5.3. Risk-Averse SDDP

The objective of a risk-averse model is to avoid risky decisions that would lead to high costs for certain unfavorable scenarios. We will consider the risk measure

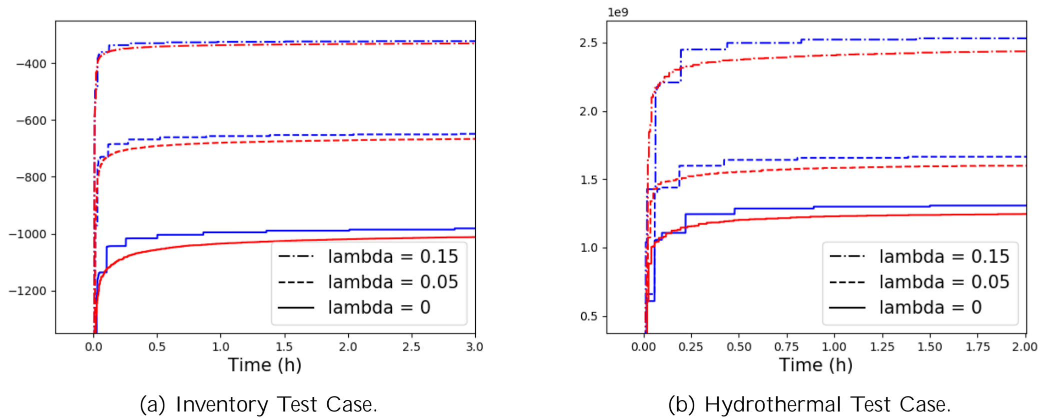
$$\rho_t[Z] = (1 - \lambda)\mathbb{E}_t[Z] + \lambda \text{AV@R}_\alpha[Z]$$

where $\lambda \in [0, 1]$ is a weighting parameter and $\text{AV@R}_\alpha[Z]$ is the Average Value-at-Risk (also called Conditional Value-at-Risk). It is defined as

$$AV@R_\alpha[Z] = V@R_\alpha[Z] + \alpha^{-1}\mathbb{E}_t[Z - V@R_\alpha[Z]]_+$$

Intuitively, $\text{AV@R}_\alpha[Z]$ is the expected value given that Z is greater than the $(1 - \alpha)$-quantile. The minimization then aims at minimizing the costs found at the tail of the distribution. As stated in Shapiro et al. (2013), the SDDP algorithm can be adapted to handle such a measure. Nevertheless, the algorithm only provides a lower bound approximation. Thus, stabilization criteria are used for deciding on convergence (Shapiro et al., 2013). We are then interested in

examining whether the BL-SDDP algorithm provides better lower bound estimates in a shorter amount of time.



(a) Inventory Test Case.

(b) Hydrothermal Test Case.

Figure 10: Risk averse BL-SDDP. The blue color corresponds to the BL-SDDP code, while the red color corresponds to SDDP. Panel (a) presents the evolution of the lower bound for the inventory test case, while panel (b) presents the hydrothermal test case.

Figure 10 presents the results when employing the risk measure introduced in section 2.3 for different choices of weighting parameters $\lambda \in [0, 1]$. Panel (a) presents the results for the inventory test case, while panel (b) presents the results for the hydrothermal test case. As we can observe, the BL-SDDP method, which corresponds to the blue color, is consistently able to achieve an improved lower bound relative to the standard SDDP method, which corresponds to the red color. In particular, we notice that, for the inventory test case, the greater improvements are for small weighing parameters, while the hydrothermal test case demonstrates considerable improvements for all the considered parameters.

5.4. Comparison of the value functions

In order to illustrate the differences between the value functions calculated by SDDP and BL-SDDP, we compare the expected value functions obtained through SDDP and BL-SDDP after running the codes for the same amount of time. The comparison is performed as follows. Let f_t, g_t denote the expected value function approximation obtained by BL-SDDP and SDDP respectively at stage t. At stage t, we consider the trial points obtained by SDDP and evaluate them at both f_t and g_t. The relative difference, for each trial point, is computed as

$$\frac{f_t(x) - g_t(x)}{\max\{|f_t(x)|, |g_t(x)|\}} \cdot 100$$

The relative difference at each stage is then averaged among the trial points. Note that this procedure should in principle lead to an advantage for SDDP, because we are using the trial points where the SDDP cuts are calculated.

Figure 11 presents the results of this procedure. Panel (a) presents the results for the inventory test case while panel (b) corresponds to the hydrothermal test case. The x axis corresponds to the stage number and the y axis corresponds to the relative difference. The

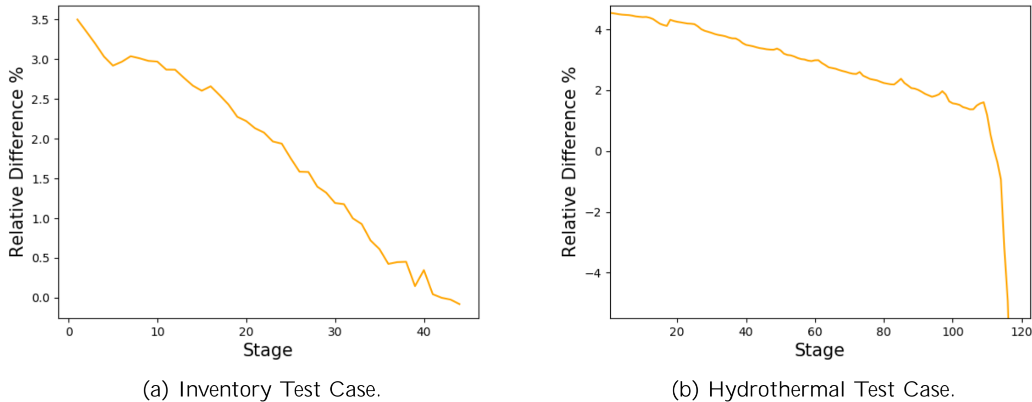


Figure 11: Comparison of expected value functions. The x axis represents the current stage. The y axis represents the relative difference between the expected value functions obtained through SDDP and BL-SDDP.

first observation is that the relative difference is a positive number for almost all stages. This indicates that the BL-SDDP algorithm is computing tighter cuts. Moreover, negative values are only encountered in few stages at the end of the time horizon. Sometimes the differences are significant. For example, in the hydrothermal test case, the relative difference in the last stage is approximately equal to 20%. However, this is expected: in the last stages the cuts produced by SDDP are tight at the trial point. Since the gap comparison is performed at the trial points obtained by SDDP, it is expected that SDDP performs better at these points (even if these points do not correspond to where an optimal policy would have "landed" at the given stage). The second observation is that the relative difference becomes greater as we move backwards in the number of stages. SDDP builds a cut for the previous stage using the current stage approximation of the expected value function approximation. A poor approximation will result in an even poorer approximation for the previous stages, thereby resulting in a back-propagation of errors. BL-SDDP is less susceptible to such an effect as the batch update results in expected value function approximations that are calculated at a large collection of trial points and therefore high-quality approximations of the expected value functions.

5.5. Comparison of BL-SDDP to PSR SDDP

The present subsection aims at comparing the performance of the BL-SDDP algorithm against the PSR SDDP commercial software. The PSR SDDP commercial software is developed for solving hydrothermal scheduling problems. We conduct our comparison on the basis of an instance of the Colombian power system that has been provided to us by PSR. The experimental results for this subsection were obtained on an Intel Core i5-6198DU CPU with 2.30GHz and 8 GB of RAM, using a single CPU.

Both codes are run with the exact same parameters. Specifically, each SDDP iteration consists of 20 samples for the forward pass. Each stage consists of 100 uncertainty realizations, namely 100 nodes per stage. Regarding the BL-SDDP algorithm, we perform batch updates every 5 SDDP iterations. The batch size corresponds to a full-batch update, namely all the experiences are used for the update. The time horizon is defined to be equal to 120 stages.

Figure 12 presents the convergence behavior for each of the two algorithms. Panel (a) presents the evolution of the lower and upper bound over iterations for the PSR software, while panel (b) presents the evolution for our BL-SDDP code. The upper bound estimate presented in panel (a) is the one reported by the PSR software. In our approach, in order to provide reliable estimates, we estimate the upper bound every 5 iterations by simulating our current policy over a large collection of samples, concretely 4000 inflow scenarios. Note that the PSR SDDP software reaches a steady state behaviour after approximately 150 iterations, at which point the difference between the lower and upper bound remains relatively constant. On the other hand, the BL-SDDP approach is able to reduce the difference between upper and lower bound significantly after each batch update.

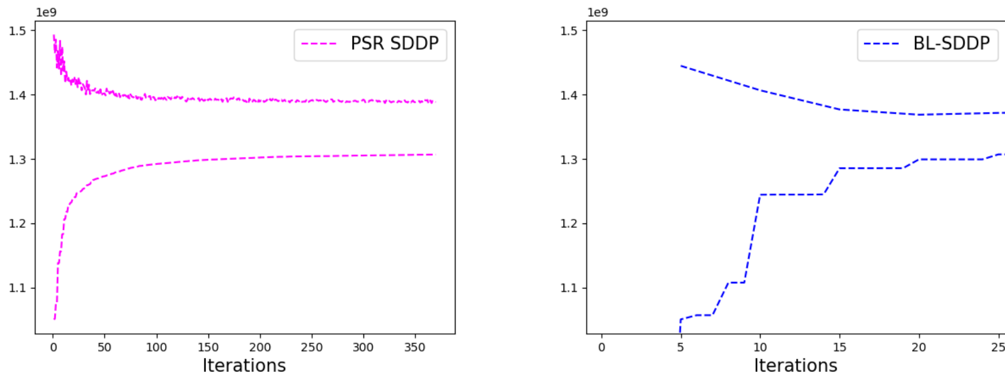


Figure 12: Lower and Upper bound evolution over iterations. Panel (a) presents the PSR SDDP software while Panel (b) presents the BL-SDDP algorithm.

Table 1 presents the total run time and the reported gap after 34 hours of run time. Note that the BL-SDDP algorithm is able to produce a better gap. The table also presents the mean cost of an out-of-sample evaluation of both policies. Concretely, 2000 out-of-sample inflow scenarios are generated using PSR software. The policies are then tested against these inflows. As we can observe, the improved gap of the BL-SDDP algorithm also results in a superior out-of-sample performance relative to the policy generated by the PSR SDDP commercial software.

	PSR SDDP	BL-SDDP
Reported Gap (%)	6.2	4.2
Time (h)	34 (1 CPU)	34 (1 CPU)
Out-Of-Sample Inflows ($)	1.437e9	1.39e9

Table 1: Comparison of the policies after 34 hours of run time.

Interestingly, such a superior performance in computing tighter optimality gaps holds even though our base SDDP implementation is considerably slower as compared to the PSR SDDP implementation. Concretely, PSR SDDP requires approximately about 30 minutes in order to compute 20 iterations while our base SDDP implementation requires approximately 3 hours in order to compute the same number of iterations.

The superior performance of the BL-SDDP algorithm in computing tighter optimality gaps, despite the less optimized performance of the subproblem solvers, can be understood in terms of the amount of "work" that each approach is performing, and in particular the number of linear programs that both approaches are solving. For ease of exposition, let us ignore the forward pass as it comprises a very small part of the computational effort. The problems solved when performing the backward pass can be described as follows:

- PSR SDDP: Evaluating a single trial point involves solving, at each stage, 100 LPs. Since the time horizon is equal to 120 stages, this amounts to a total of $119 \quad 100$ problems. Every iteration considers 20 forward samples. Thus, 20 trial points are generated per iteration. Consequently, a total of $100 \quad 119 \quad 20$ problems are solved per iteration. Over 370 iterations, this amounts to a total of $100 \quad 119 \quad 20 \quad 370 = 88,060,000$ problems.

- BL-SDDP: The algorithm performs a total of 25 usual SDDP iterations. As explained in the previous paragraph, this results in $100 \quad 119 \quad 20 \quad 25 = 5950000$ problems after the execution of the 25 usual SDDP iterations. In this case, we have to add the problems that are solved when performing the full-batch updates. After N iterations, a total of $N \ 20$ trial points need to be solved. As we mention previously, for each trial point a total of $119 \ 100$ LPs need to be solved. Thus, performing a full-batch update after N iterations would require solving $119 \ 100 \ 20 \ N$ LPs. As the full-batch update is performed every 5 iterations, this means $N = 5, 10, 15, 20, 25$. Thus, the full-batch update step throughout the entire execution of the algorithm requires solving $100 \quad 119 \quad 20 \quad (5 + 10 + 15 + 25) = 17,850,000$ problems. Adding the full-batch update LPs and the LPs of the usual SDDP iterations results in a total of $23,800,000$ problems, which represents 27% of the problems that the PSR software is solving. In short, the difference is due to the fact that the BL-SDDP algorithm avoids over-exploring and thus avoids redundant computation in extra iterations.

As a consequence of the aforementioned observation, we additionally note that the expected value function approximations for the BL-SDDP algorithm are considerably lighter. Concretely, PSR SDDP performs 370 iterations before terminating, thus the expected value function approximation consists of approximately $370 \quad 20 = 7,400$ cuts per stage. Instead, the BL-SDDP code requires approximately $25 \quad 20 = 500$ cuts per stage.

6. Conclusions

In this paper we propose a novel variant of the SDDP algorithm by introducing ideas from the reinforcement learning framework to the multistage stochastic framework. This SDDP variant, which we refer to as *Batch Learning SDDP*, is shown to improve the convergence behavior of the SDDP algorithm. The algorithm is evaluated against an inventory management problem and a realistic instance of hydrothermal scheduling in Colombia. The main observations of the paper are summarized below, in the form of the following conclusions.

(i) We propose a description of SDDP as a reinforcement learning technique and (ii) propose an algorithm that integrates ideas of reinforcement learning into the SDDP framework. (iii) The BL-SDDP algorithm converges faster than the commercial PSR SDDP software. (iv) The BL-SDDP parallel scheme has a positive parallel efficiency performance. (v) The proposed BL-SDDP algorithm can handle different risk measures such as risk neutral and risk averse formulations. (vi) The BL-SDDP algorithm narrows the optimality gap of the tested instances by relying on fewer iterations than SDDP and thus provides lighter descriptions of the value functions.

In our work, we demonstrate how the combination of reinforcement learning ideas and the standard SDDP algorithm can lead to a novel scheme with superior convergence for a variety of case studies. This opens a research path for investigating whether other reinforcement learning techniques can be integrated into the SDDP framework. In particular, strategies for selecting a proper batch could prove to be of great help for the performance of the algorithm. Furthermore, we have proposed a parallel variant with positive results. Nevertheless, as the algorithm is synchronous, it remains susceptible to well-known synchronization issues (Bertsekas & Tsitsiklis, 1989). This motivates the question of how to improve the scheme so as to avoid synchronization bottlenecks and thus improve the usage of computational resources. These questions will be investigated in future research.

Acknowledgements

Computational resources have been provided by the Consortium des Équipements de Calcul Intensif (CÉCI), funded by the Fonds de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under Grant No. 2.5020.11 and by the Walloon Region. The first and second author have been supported financially by the Bauchau family in the context of the Bauchau prize which is administed by UCLouvain. This work was supported in part by the European Commission's Horizon 2020 Framework Program under grant agreement No 864537 (FEVER Project). The authors would also like to thank Joaquim Garcia, from PSR, for providing the hydrothermal test case which is considered in this paper, as well as to Mauricio Junca, from los Andes university in Colombia, for his valuable input.

References

Asamov, T., & Powell, W. B. (2018). Regularized decomposition of high-dimensional multistage stochastic programs with markov uncertainty. *SIAM Journal on Optimization*, *28*, 575–595.

Bertsekas, D. P., & Tsitsiklis, J. N. (1989). *Parallel and distributed computation: numerical methods* volume 23. Prentice hall Englewood Cliffs, NJ.

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM review*, *59*, 65–98. URL: https://doi.org/10.1137/141000671.

De Matos, V., Philpott, A. B., Finardi, E. C., & Guan, Z. (2010). *Solving long-term hydro-thermal scheduling problems*. Technical Report Technical report, Electric Power Optimization Centre, University of Auckland.

De Matos, V. L., Philpott, A. B., & Finardi, E. C. (2015). Improving the performance of stochastic dual dynamic programming. *Journal of Computational and Applied Mathematics*, *290*, 196–208.

Dowson, O., & Kapelevich, L. (2017). SDDP.jl: a Julia package for stochastic dual dynamic programming. *Optimization Online*, . URL: http://www.optimization-online.org/DB_HTML/2017/12/6388.html.

Dowson, O., Philpott, A., Mason, A., & Downward, A. (2019). A multi-stage stochastic optimization model of a pastoral dairy farm. *European Journal of Operational Research*, *274*, 1077–1089.

Dunning, I., Huchette, J., & Lubin, M. (2017). Jump: A modeling language for mathematical optimization. *SIAM Review*, *59*, 295–320. doi:10.1137/15M1020575.

Flach, B., Barroso, L., & Pereira, M. (2010). Long-term optimal allocation of hydro generation for a price-maker company in a competitive market: latest developments and a stochastic dual dynamic programming approach. *IET generation, transmission & distribution*, *4*, 299–314.

Guigues, V. (2017). Dual dynamic programing with cut selection: Convergence proof and numerical experiments. *European Journal of Operational Research*, *258*, 47–57.

Guigues, V., & Bandarra, M. (2019). Single cut and multicut sddp with cut selection for multistage stochastic linear programs: convergence proof and numerical experiments. *arXiv preprint arXiv:1902.06757*, .

Helseth, A., & Braaten, H. (2015). Efficient parallelization of the stochastic dual dynamic programming algorithm applied to hydropower scheduling. *Energies*, *8*, 14287–14297.

Kalyanakrishnan, S., & Stone, P. (2007). Batch reinforcement learning in a complex domain. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems* (pp. 1–8).

Lange, S., Gabel, T., & Riedmiller, M. (2012). Batch reinforcement learning. In *Reinforcement learning* (pp. 45–73). Springer.

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, *8*, 293–321.

Löhndorf, N., & Shapiro, A. (2019). Modeling time-dependent randomness in stochastic dual dynamic programming. *European Journal of Operational Research*, *273*, 650–661.

Löhndorf, N., & Wozabal, D. (2020). Gas storage valuation in incomplete markets. *European Journal of Operational Research*, .

Löhndorf, N., Wozabal, D., & Minner, S. (2013). Optimizing trading decisions for hydro storage systems using approximate dual dynamic programming. *Operations Research*, *61*, 810–823.

Machado, F. D., Diniz, A. L., Borges, C. L., & Brandão, L. C. (2021). Asynchronous parallel stochastic dual dynamic programming applied to hydrothermal generation planning. *Electric Power Systems Research*, *191*, 106907.

Mnih, V., Kavukcuoglu, K., & Silver, e. a. (2015). Human-level control through deep reinforcement learning. *nature*, *518*, 529–533.

Pereira, M. V., & Pinto, L. M. (1991). Multi-stage stochastic optimization applied to energy planning. *Mathematical programming*, *52*, 359–375.

Pieters, M., & Wiering, M. A. (2016). Q-learning with experience replay in a dynamic environment. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)* (pp. 1–8). IEEE.

Pinto, R. J., Borges, C. T., & Maceira, M. E. (2013). An efficient parallel algorithm for large scale hydrothermal system operation planning. *IEEE Transactions on Power Systems*, *28*, 4888–4896.

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

Puterman, M. L. (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

Shapiro, A., Dentcheva, D., & Ruszczyński, A. (2014). *Lectures on stochastic programming: modeling and theory*. SIAM.

Shapiro, A., Tekaya, W., da Costa, J. P., & Soares, M. P. (2013). Risk neutral and risk averse stochastic dual dynamic programming method. *European journal of operational research*, *224*, 375–391.

da Silva, E. L., & Finardi, E. C. (2003). Parallel processing applied to the planning of hydrothermal systems. *IEEE Transactions on Parallel and Distributed Systems*, *14*, 721–729.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Appendix

Proof of lemmas

Lemma 2.2. *The problems MDP-P and MSP-P are equivalent.*

Proof. Let's consider a feasible solution of problem MSP-P. Furthermore, we can ask such a solution to be a vertex of the polyhedrons defining the linear programs. Then for every t and $\xi_t \; 2 \; \Omega_t$ we have feasible values $x_t(\xi_t), y_t(\xi_t)$ and an objective cost

$$u_1^T x_1 + v_1 y_1 + \mathsf{E}\Big[u_2^T x_2 + v_2^T y_2 + \mathsf{E}\Big[\quad + \mathsf{E}\Big[u_T^T x_T + v_T^T y_T\Big]\Big]\Big]$$

Note that we can define a policy π_t such that for $s_t = (x_{t-1}(\xi_{t-1}), \xi_t)$ we have $\pi_t(s_t) = (x_t(\xi_t), y_t(\xi_t))$. Moreover, under such a policy,

$$\mathsf{E}\Big[\sum_{t=1}^{T} C_t(s_t, a_t)\Big] = u_1^T x_1 + v_1 y_1 + \mathsf{E}\Big[u_2^T x_2 + v_2^T y_2 + \mathsf{E}\Big[\quad + \mathsf{E}\Big[u_T^T x_T + v_T^T y_T\Big]\Big]\Big]$$

Therefore, every vertex solution of MSP-P gives a policy π such that when evaluated in MDP-P gives the same cost. Thus, MDP-P MSP-P. Similarly, given any policy π we can define a feasible solution for MSP-P such that when evaluated gives the same result as when evaluating the policy, and so MDP-P MSP-P. □

Supporting hyperplane update rule

Given an action $a_t^n = (x_t^n, y_t^n)$ let's consider the update rule $Q_t^n = U(a_t^n, Q_{t+1}^n)$ defined by

$$Q_t^n(s_t, a_t) = C_t(s_t, a_t) + \max_{i=1; \; ;n} \Big\{a_i + b_i x_t\Big\}$$

Where a_i, b_i are some coefficients that depend on x_t^i and on the next time step Q-factor Q_{t+1}^n. Concretely, the update rule can be described as follows.

1. Input: $a_t^n = (x_t^n, y_t^n)$ and Q_{t+1}^n
2. For $\xi_{t+1} \; 2 \; \Omega_{t+1}$
   (1.1) Consider the state $s_{t+1}^n = (x_t^n, \xi_{t+1})$. Solve the linear program

$$V_{t+1}^n(s_{t+1}^n) = \min_{a_{t+1} \in \mathcal{A}(s_{t+1})} Q_{t+1}^n(s_{t+1}^n, a_{t+1})$$

   (1.2) Get the dual multiplier $\pi_t^n(\xi_{t+1})$ and compute the quantity

$$g_{t+1}^n(\xi_{t+1}) = \pi_t^n(\xi_{t+1}) \quad B_{t+1}(\xi_{t+1})$$

3. The Q-factors are updated as,

$$Q_t^n(s_t, a_t) = C_t(s_t, a_t) + \max_{i=1; \; ;n} \Big\{ \sum_{t+1 \in \quad t+1} P(\xi_{t+1} j \xi_t) \quad \Big(V_{t+1}^i(s_{t+1}^i) + g_{t+1}^i(\xi_{t+1})(x_t^i \quad x_t)\Big)\Big\}$$

Further details can be found in Shapiro et al. (2013, 2014); Pereira & Pinto (1991).

Formulation of the Inventory Control Problem

In this section we present the model of the inventory control problem with lead times and lost sales. Given T stages, and assuming uncertain demand, which is assumed to follow a Markov Chain, the problem can be modeled as a multistage stochastic problem. We denote the expected value function as $V_{t+1}(z_t, \xi_t)$. We then compute the value function $V_t(z_{t-1}, \xi_t)$ by solving the following problem each stage:

$$
\begin{aligned}
V_t(z_{t-1}, \xi_t) = \max \sum_{n \in \mathcal{N}} P \quad s_{t;n} \quad HC \quad v_{t;n} \quad PC \quad x_{t;n} + V_{t+1}(z_t, \xi_t) & \\
s.t.\ v_{t;n} = v_{t-1;n} + x_{t-L;n} \quad s_{t;n} & \qquad n\ 2\ N \\
s_{t;n} \quad v_{t-1;n} & \qquad n\ 2\ N \\
s_{t;n} \quad D_{t;n}(\xi_t) & \qquad n\ 2\ N \\
x_{t;n} \quad C & \qquad n\ 2\ N \\
v_{t;n}, s_{t;n}, x_{t;n} \quad 0 & \qquad n\ 2\ N
\end{aligned}
$$

The variables can be described as follows:

- $z_{t;n}$: The state variable. This is a vector containing the on-hand inventory $v_{t;n}$ for product $n\ 2\ N$, as well as the ordered quantities $x_{t-l;n}$ for $l = 1, ..., L$ with L being the lead time.
- $s_{t;n}$: Variable representing the amount of sold items for product $n\ 2\ N$.

The parameters can be described as follows:

- L: The considered lead time.
- P: The sales price.
- HC: The inventory holding cost.
- PC: the purchase cost.
- $D_{t;n}$: The demand for product $n\ 2\ N$.
- C: The buying capacity.

The sets are defined as:

- N: The set of possible products.

The instance that we consider is defined for 1 product and a lead time of 5 periods, thereby producing a 6-dimensional state space. The considered time horizon is equal to 50 stages. We consider 100 realizations of uncertainty at each stage.

Hydrothermal Scheduling Problem

The problem aims at minimizing the operational costs of thermal plants and curtailed demand by determining optimal water levels in the hydro reservoirs. Let $V_{t+1}(v_t, \xi_t)$ denoted the expected value function. Each stage is described in terms of the value function $V_t(v_{t-1}, \xi_t)$, which is defined by solving the following problem:

$$
\begin{aligned}
V_t(v_{t-1}, \xi_t) = \min \; & \sum_{n \in \mathcal{G}} C_n \quad g_{n;t} + \text{VOLL} \quad ls_t + V_{t+1}(v_t, \xi_t) \\
s.t. \; & \sum_{n \in \mathcal{H}} P_n \quad q_{t;n} + \sum_{n \in \mathcal{G}} g_{t;n} + ls_t = L_t \\
& v_t = v_{t-1} + A_t(\xi_t) + M(q_t + s_t) \\
& g_t \quad \bar{G} \\
& v_t \quad \bar{V} \\
& q_t \quad \bar{Q} \\
& g_t, v_t, q_t, s_t \quad 0
\end{aligned}
$$

The variables are described as follows:

- $v_{t;n}$: The state variable. It represents the storage level of reservoir $n \; 2 \; H$.
- $q_{t;n}$: Variable representing the water turbined outflow volume of reservoir $n \; 2 \; H$.
- $s_{t;n}$: Variable which accounts for the spilled volume of water at reservoir $n \; 2 \; H$.
- $g_{t;n}$: The vector of generated power from thermal plants $n \; 2 \; G$.
- ls_t: Variable which accounts for load shedding.

The parameters are given as follows:

- C_n: The generation cost of thermal plant $n \; 2 \; G$.
- P_n: The energy generation coefficient for the turbined outflow for hydro plant $n \; 2 \; H$.
- L_t: Load at stage t.
- A_t: The inflow vector.
- M : Matrix representing the hydrological topology.
- $\bar{G}, \bar{V}, \bar{Q}$: Upper limits on the variables.

The sets are described as follows:

- H: The set of reservoirs.
- G : The set of thermal plants.