On Inferring Cumulative Constraints

Konstantin Sidorov
Delft University of Technology, The Netherlands

Abstract

Cumulative constraints are central in scheduling with constraint programming, yet propagation is typically performed per constraint, missing multi-resource interactions and causing severe slowdowns on some benchmarks. I present a preprocessing method for inferring additional cumulative constraints that capture such interactions without search-time probing.

This approach interprets cumulative constraints as linear inequalities over occupancy vectors and generates valid inequalities by (i) discovering covers, the sets of tasks that cannot run in parallel, (ii) strengthening the cover inequalities for the discovered sets with lifting, and (iii) injecting the resulting constraints back into the scheduling problem instance. Experiments on standard RCPSP and RCPSP/max test suites show that these inferred constraints improve search performance and tighten objective bounds on favorable instances, while exhibiting more stable performance on unfavorable instances than the prior work. Additionally, these experiments discover 25 new lower bounds and five new best solutions; seven of the lower bounds are obtained directly from the inferred constraints.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Discrete optimization; Mathematics of computing → Integer programming

Keywords and phrases Constraint programming, scheduling, cumulative, lifting, valid inequalities

Digital Object Identifier 10.4230/LIPIcs.CP.2026.1

Related Version *Preprint*: `https://arxiv.org/abs/2602.15635`

Supplementary Material The experimental results reported in this paper are obtained with:
Software (Lifting implementation and infrastructure): `https://doi.org/10.5281/zenodo.20044686`
Dataset (Solver logs and report files): `https://doi.org/10.5281/zenodo.18663551`

Funding *Konstantin Sidorov*: supported by the TU Delft AI Labs program as part of the XAIT lab.

Acknowledgements I would like to thank Imko Marijnissen for the enlightening discussions that sparked the development of the technique presented in this paper, and the anonymous reviewers for improving the presentation of the approach and empirical results.

1 Introduction

Cumulative constraints and their use in scheduling problems are one of the longstanding success stories of constraint programming (CP), serving as a central abstraction and a source of powerful inferences about resource-constrained scheduling problems. A large body of work has demonstrated how powerful propagation on a single cumulative resource can be, including effective detection of overload situations, pruning based on resource envelopes, and increasingly sophisticated explanations and propagator combinations. All these developments have contributed to the state-of-the-art performance of CP solvers on scheduling benchmarks.

Nevertheless, most existing techniques in this field reason about one cumulative constraint at a time. Even when a model contains several resources, propagation is typically performed independently per resource, with indirect interactions only mediated through domain narrowing of start-time variables. However, the recent work by Sidorov et al. [33] has made a first step to expose the limitations of this single-resource viewpoint, including the $3n$ *problem* structure, a pathological case when an instance is equivalent to a trivially infeasible

single-machine scheduling problem but is intractable for a conventional lazy clause-generating solver, a paradigm subsuming many of the state-of-the-art CP codes.

While Sidorov et al. have addressed a special (disjunctive) case of this problem with a *Unite and Lead* (referred to in the following as UnL) approach and managed to discover large hidden structures in several known research benchmarks, their technique has two fundamental limitations. First, while it succeeds at discovering disjunctive structures (that is, the collections of tasks that cannot be run in parallel), it does not support any extensibility for deriving more general multi-resource interactions. Second, UnL has to explicitly probe for disjunctive cliques throughout the search, incurring significant overhead even in instances where no such structure exists. Since the clique search overhead is not offset by stronger reasoning on instances without a disjunctive structure, the solver performance deteriorates sharply on such problems. As a result, Sidorov et al. show that, when UnL does not improve the search, it commonly slows down the solver by more than an order of magnitude.

In this paper, I address these limitations by proposing a new algorithmic framework for inferring cumulative constraints at the root search node. Instead of restricting attention to disjunctive reasoning, this approach operates over cumulative constraints as *linear inequalities* on a special representation of start time variables (occupancy vectors), and reasoning across multiple resources reduces to *generating valid cutting planes* in the same sense as it is understood in integer programming. This viewpoint subsumes a wide range of multi-constraint reasoning techniques from earlier work, including the version of UnL restricted to the root node and classical lower-bound techniques for RCPSPs.

Next, I leverage this connection by describing a workflow for inferring cumulative constraints based on the idea of *lifting* [27]. Lifting is a general-purpose technique from mixed-integer programming for tightening the generated cutting planes $\boldsymbol{a}^T\boldsymbol{x} \leq b$ by picking a variable y not mentioned in the inequality and finding the largest β such that $\boldsymbol{a}^T\boldsymbol{x} + \beta y \leq b$ is still valid. I apply this idea by (i) discovering sets of tasks C that cannot be run in parallel, (ii) lifting the constraint of the form "tasks in C cannot consume more than $\big(|C| - 1\big)$ units of resource," and (iii) introducing the lifted constraints as new cumulative constraints into the model.

I evaluate the method as a preprocessing step for two state-of-the-art CP solvers on a suite of standard RCPSP benchmarks. The results suggest that the approach described in this paper successfully retains most of the positive traits of UnL (substantial gains on instances with hidden structure), but does so with less severe runtime penalties than UnL. This is helpful for advancing the best-known bounds for the benchmark instances: with the presented approach, I report the improved schedules for five benchmarks and tighten the lower bounds for 25 instances, with seven of them not requiring any search to verify the correctness of the new bound.

The remainder of the paper is organized as follows. Section 2 introduces constraint programming, the `Cumulative` constraint, and the multi-resource scheduling model that motivate this work. Section 3 establishes the theoretical foundation of the approach: it reformulates `Cumulative` constraints as linear inequalities over occupancy vectors, recalls the necessary machinery of covers and lifting from integer programming, and connects valid inequalities of the occupancy-vector polyhedron to new `Cumulative` constraints via *inference transfer lemma*. Section 4 builds on this foundation by describing the complete lifting-based procedure for inferring auxiliary `Cumulative` constraints, including cover enumeration, the lifting workflow, and the quality metric used to select the most restrictive inferred constraints. I evaluate this approach in Section 5 from two angles: (a) as a preprocessing technique for adding cumulative constraints to the input instance before running a CP solver, and (b) as a

technique for inferring `Cumulative` constraints with unit capacity,[1] by comparing it with the UnL approach. Section 6 discusses earlier methods for joint reasoning on cumulative constraints and their connection with the lifting approach. Finally, Section 7 concludes the narrative and discusses future research directions.

2 Scheduling with `Cumulative` constraints

Constraint programming and the `Cumulative` constraint. A constraint satisfaction problem (CSP) consists of a tuple $(\mathcal{X}, \mathcal{C}, \mathcal{D})$ where $\mathcal{X}$ is the set of *variables*, $\mathcal{C}$ is the set of *constraints* which specify the relations between variables, and $\mathcal{D}$ is the *domain* which specifies for each variable which values it can take [30]. A *solution* $\mathcal{I}$ is a mapping that maps each variable in $\mathcal{X}$ to a single value in the domain of that variable in $\mathcal{D}$ which satisfies all of the constraints in $\mathcal{C}$.

Constraint programming (CP) is a paradigm for solving CSPs; CP solvers enforce constraints through *propagators*, the functions that each enforce the local consistency of one constraint by removing values from $\mathcal{D}$ that cannot participate in any solution. CP solvers interleave propagation with search via a branch-and-bound procedure: after applying the propagators, the solver makes a decision that creates several subproblems by splitting the domain of a variable into two or more parts. This process of applying propagators and making decisions is performed until either a solution $\mathcal{I}$ is found, the problem is found to be unsatisfiable, or a termination criterion is met.

`Cumulative` is a constraint useful for many scheduling problems to model limited renewable resources, such as available work-hours. Definition 1 formalizes the notion of a task and the `Cumulative` constraint, where the variable s_j encodes the start time of task j, and $(s_j + d_j)$ encodes its finish time. Thus, we implicitly associate each task $j \in T$ with an interval $[s_j, s_j + d_j)$.

Throughout the text, I use the Iverson bracket $[\![P]\!]$, defined for an arbitrary predicate P as follows: $[\![P]\!] = 1$, if P is true, and $[\![P]\!] = 0$ otherwise.

▶ **Definition 1.** *Let T be a set of **tasks**, and let a task $j \in T$ be defined by its start variable s_j, resource usage $a_j \in \mathbb{Z}_{\geq 0}$, and duration $d_j \in \mathbb{Z}_{\geq 0}$. Finally, let $b \in \mathbb{Z}_{\geq 0}$ be the **capacity** of the resource. Then the* `Cumulative`$(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{d}, b)$ *constraint is the condition that at any time point τ the resource usage of intervals $[s_j, s_j + d_j)$ covering τ does not exceed the capacity:*

$$\forall \tau \in \mathbb{Z} \ : \ \sum_{j \in T} a_j [\![s_j \leq \tau < s_j + d_j]\!] \leq b. \tag{1}$$

An important special case of `Cumulative` is where the resource has unit capacity $b = 1$ and each task has resource usage $a_j \in \{0, 1\}$, known as `Disjunctive`. Note that $a_j = 0$ is permitted and represents a task that does not consume the resource modeled by this constraint, which is helpful for a multi-resource model, since tasks do not have to consume all the modeled resources.

This special case is significant because many inference procedures that are intractable for `Cumulative` constraints can be efficiently executed for `Disjunctive` constraints. For example, determining satisfiability of a single `Cumulative` constraint is NP-complete [2], but determining satisfiability of a single `Disjunctive` constraint can be done in polynomial time [37, Section 3].

[1] Also known as disjunctive or no-overlap constraints.

Scheduling problems with multiple `Cumulative` constraints Many practical scheduling problems require reasoning about several resources simultaneously. A canonical example is the *resource-constrained project scheduling problem* (RCPSP) [8], where a set of n tasks T must be scheduled—that is, each task j has to get a starting time s_j—subject to m renewable resources and precedence constraints $s_k - s_j \geq d_j$ for some pairs of tasks (j, k). Each task $j \in T$ has a fixed duration $d_j \in \mathbb{Z}_{\geq 0}$ and consumes $a_{ij} \in \mathbb{Z}_{\geq 0}$ units of resource i while active. Each resource i has a capacity $b_i \in \mathbb{Z}_{\geq 0}$, and the objective is to minimize the makespan – the time elapsed from the earliest task start[2] to the latest task finish, $\max_{j \in T}(s_j + d_j)$. The RCPSP/max variant additionally admits difference constraints of the form $s_k - s_j \geq \gamma_{jk}$ between pairs of tasks, generalizing precedence constraints.

A common trait of such problems is that their CP models contain the following collection of m `Cumulative` constraints over n variables $\boldsymbol{s}$ encoding the start times of tasks in T:

$$\texttt{Cumulative}(\boldsymbol{s}, \boldsymbol{a}_i, \boldsymbol{d}, b_i) \quad (i = 1, \ldots, m). \tag{2}$$

This part of the CP model is defined by the duration vector $\boldsymbol{d} \in \mathbb{Z}^n_{\geq 0}$ shared between constraints, the capacity vector $\boldsymbol{b} \in \mathbb{Z}^m_{\geq 0}$ containing the capacity of each `Cumulative`, with b_i being the capacity of the i-th constraint, and the resource usage matrix $A \in \mathbb{Z}^{m \times n}_{\geq 0}$, where the i-th row $\boldsymbol{a}_i$ contains the consumptions on the i-th constraint and the j-th column contains the consumptions of the j-th task on all constraints.

In a conventional CP solver, each of these constraints is handled by an independent propagator, with interactions between resources mediated only indirectly through domain narrowing of the start-time variables s. On the other hand, the joint modeling of resource constraints offers a richer inference structure, as stronger conclusions can be drawn by reasoning collectively; the relevant ideas in the related work are discussed in Section 6.

3 `Cumulative` constraints as linear inequalities

This section is organized as follows. Subsection 3.1 introduces occupancy vectors and establishes that a collection of `Cumulative` constraints can be restated as a single matrix inequality over those vectors – this reformulation is the key conceptual bridge between CP and integer programming that the rest of the paper builds on. Subsection 3.2 recalls the notions of covers, cover inequalities, and lifting from the theory of knapsack polyhedra [27, 16]; these are standard tools from integer programming presented here for completeness. Finally, Subsection 3.3 states and proves the *inference transfer lemma* – the main theoretical contribution of this paper – which establishes that any valid inequality of the occupancy-vector polyhedron can be introduced into the CP model as a new `Cumulative` constraint, and illustrates the complete workflow on an example.

3.1 Polyhedral view via occupancy vectors

We start by introducing a representation of the start-time variables that we need in order to reason about linear inequalities. Intuitively, we change the variable representation from an integer start time s_j to a binary vector that records at each integer time point whether a task j is active.

[2] We assume without loss of generality that tasks are available from time 0, and therefore $\min_j s_j = 0$.

▶ **Definition 2.** *Given a variable x of a CSP and an integer d, from now on referred to as* ***time*** *and* ***duration****, the* ***occupancy vector*** *$\langle x \mid d \rangle$ is a binary vector indexed by integer time points $t \in \mathbb{Z}$ with a finite support of d consecutive ones: $\langle x \mid d \rangle_t := [\![x \le t < x + d]\!]$.*

▶ Note 3. The occupancy vector $\langle x \mid d \rangle$ is structurally identical to the $\texttt{pulse}(x, x + d, 1)$ function of CP Optimizer [21]: both encode the presence of a unit-height pulse over $[x, x + d)$. The key difference is that `pulse` is used as a building block for cumulative resource expressions evaluated at a single time point, whereas $\langle x \mid d \rangle$ is treated as a vector admitting linear-algebraic operations, in particular, the matrix-vector product in Lemma 4.

The key point behind this definition is that a system of `Cumulative` constraints from Equation 2 can be restated as a system of *linear* inequalities over occupancy vectors:

▶ **Lemma 4** (polyhedral view)**.** *An assignment of n variables $\boldsymbol{s}$ satisfies m constraints $\texttt{Cumulative}(\boldsymbol{s}, \boldsymbol{a}_i, \boldsymbol{d}, b_i)$ for $i = 1, \dots, m$ if and only if the occupancy vectors of $\boldsymbol{s}$ satisfy*

$$A \begin{pmatrix} \langle s_1 \mid d_1 \rangle \\ \vdots \\ \langle s_n \mid d_n \rangle \end{pmatrix} \le \begin{pmatrix} \boldsymbol{b} & \cdots & \boldsymbol{b} \end{pmatrix}, \tag{3}$$

where the right-hand side denotes the matrix with every column equal to $\boldsymbol{b}$, indexed by time points $t \in \mathbb{Z}$.

Proof. ($\Rightarrow$) Observe that the elements of the left-hand side and right-hand side matrices in row r and column t are equal to $\sum_j a_{rj} \langle s_j \mid d_j \rangle_t$ and b_r respectively. Consider an arbitrary pair of indices r, t; we need to show that $\sum_j a_{rj} \langle s_j \mid d_j \rangle_t \le b_r$, and by Definition 2, this is equivalent to

$$\sum_j a_{rj} [\![s_j \le t < s_j + d_j]\!] \le b_r.$$

But this is implied by Definition 1 for the constraint $\texttt{Cumulative}(\boldsymbol{s}, \boldsymbol{a}_r, \boldsymbol{d}, b_r)$, which is satisfied by the lemma statement.

($\Leftarrow$) To show that $\texttt{Cumulative}(\boldsymbol{s}, \boldsymbol{a}_r, \boldsymbol{d}, b_r)$ holds for any r given Equation 3, take an arbitrary time point t and write Equation 1 for it as $\sum_j a_{rj} [\![s_j \le t < s_j + d_j]\!] \le b_r$. However, by definition of the matrix product, this is exactly the inequality between the elements of matrices in Equation 3 at row r, column t. ◀

This reformulation is similar to the time-resource decomposition [31] in the sense that it linearizes a `Cumulative` constraint, but the main difference is that Equation 3 introduces *one matrix* inequality, which is possible through the use of occupancy vectors. Conversely, the time-difference decomposition introduces one constraint per time point, as well as a collection of constraints that enforce Definition 2. While using Equation 3 directly does not provide any advantage for solving CSPs with `Cumulative` constraints, it provides a new viewpoint for such models: they can be seen as models with *polyhedral* constraints over occupancy vectors, with the coefficients of inequalities mapping one-to-one to the arguments of `Cumulative` constraints.

Crucially for this work, this reformulation raises the following question: what *other* linear inequalities are valid for any occupancy vectors satisfying Equation 3? The following section introduces the concepts of polyhedral geometry that we need to answer this question.

3.2 Covers and lifting

The occupancy vectors satisfying Equation 3 form a set of the form $\mathcal{P}(A, \boldsymbol{b}) := \{\boldsymbol{x} \in \{0,1\}^n : A\boldsymbol{x} \leq \boldsymbol{b}\}$; we now develop the tools needed to reason about valid inequalities for such sets. In general, A and $\boldsymbol{b}$ do not contain *all* inequalities that hold on $\mathcal{P}(A, \boldsymbol{b})$; this gap is filled by the following definition:

▶ **Definition 5.** *An inequality $\boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0$, also written as $(\boldsymbol{\pi}, \pi_0) \in \mathbb{R}^{n+1}_{\geq 0}$, is called a **valid inequality** for $\mathcal{P}(A, \boldsymbol{b})$ if it holds for any point in $\mathcal{P}(A, \boldsymbol{b})$.*

Discovering valid inequalities of a given polyhedron $\mathcal{P}(A, \boldsymbol{b})$ is itself a theoretically rich topic with major practical implications for integer programming solvers. For the purposes of this work, we only need a few core notions; I refer the interested reader to survey the literature on knapsack polyhedra [16] for a more systematic exposition. Consider first a simple family of valid inequalities:

▶ **Definition 6.** *Given a polyhedron $\mathcal{P}(A, b)$ with a single constraint $\boldsymbol{a}^T\boldsymbol{x} \leq b$, we say that $C \subseteq \{1, \ldots, n\}$ is a **cover** if $\sum_{i \in C} a_i > b$.*

▶ **Proposition 7.** *A **cover inequality** $\sum_{i \in C} x_i \leq |C| - 1$ is a valid inequality for $\mathcal{P}(A, b)$ if C is a cover.*

Next, we need a general-purpose procedure for increasing the left-hand side of a valid inequality $\boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0$—which may or may not be a cover inequality—known as *lifting* [27]. This procedure maintains the set C of variables that have already been considered for adding to the inequality, starting with $C \leftarrow \{i : \pi_i \neq 0\}$, and repeating the following steps until $C = \{1, \ldots, n\}$:

1. Choose an unused variable $i \in \{1, \ldots, n\} \setminus C$ or terminate if none exist.
2. Solve an auxiliary subproblem over the variables in C

$$v^*(\boldsymbol{\pi}, \pi_0, C, i; A, \boldsymbol{b}) := \begin{array}{ll} \text{maximize} & \displaystyle\sum_{c \in C} \pi_c x_c \\ \boldsymbol{x} \in \{0,1\}^{|C|} & \\ \text{subject to} & \displaystyle\sum_{c \in C} a_{j,c} x_c \leq b_j - a_{j,i} \quad (j = 1, \ldots, m) \end{array} \tag{4}$$

   that corresponds to maximizing the left-hand side of the current inequality $\boldsymbol{\pi}^T\boldsymbol{x}$ when the variable x_i under consideration is set to 1.
3. Set $\pi_i \leftarrow \pi_0 - v^*(\boldsymbol{\pi}, \pi_0, C, i; A, \boldsymbol{b})$ and $C \leftarrow C \cup \{i\}$, where $v^*(\boldsymbol{\pi}, \pi_0, C, i; A, \boldsymbol{b})$ is the maximum value of the objective of the problem given by Equation 4.

The lifting machinery applies to any polyhedron $\mathcal{P}(A, \boldsymbol{b})$ with nonnegative integer coefficients. In particular, it applies to the occupancy-vector polyhedron arising from Equation 3 – and crucially, any valid inequality discovered for that polyhedron can be translated directly back into a `Cumulative` constraint. The next section states this correspondence formally.

3.3 From valid inequalities to `Cumulative` constraints

First, we prove the main result of Section 3 that connects the scheduling problems with `Cumulative` constraints with the theory of valid inequalities:

▶ **Lemma 8** (inference transfer). *Any assignment of n variables $\boldsymbol{s}$ satisfying m constraints $\texttt{Cumulative}(\boldsymbol{s}, \boldsymbol{a}_i, \boldsymbol{d}, b_i)$ for $i = 1, \ldots, m$ also satisfies $\texttt{Cumulative}(\boldsymbol{s}, \boldsymbol{\pi}, \boldsymbol{d}, \pi_0)$ as long as $(\boldsymbol{\pi}, \pi_0)$ is a valid inequality for the polyhedron $\mathcal{P}(A, \boldsymbol{b})$.*

Proof. We start by applying Lemma 4 to rewrite the implication we need to prove in a polyhedral view: $AS \leq \begin{pmatrix} \boldsymbol{b} & \cdots & \boldsymbol{b} \end{pmatrix} \implies \boldsymbol{\pi}^T S \leq \begin{pmatrix} \pi_0 & \cdots & \pi_0 \end{pmatrix}$, where

$$S := \begin{pmatrix} \langle s_1 \mid d_1 \rangle \\ \vdots \\ \langle s_n \mid d_n \rangle \end{pmatrix}.$$

If this implication is not true, then there is a satisfying assignment of variables $\boldsymbol{s}$ such that $AS \leq \begin{pmatrix} \boldsymbol{b} & \cdots & \boldsymbol{b} \end{pmatrix}$ holds but $\boldsymbol{\pi}^T S \leq \begin{pmatrix} \pi_0 & \cdots & \pi_0 \end{pmatrix}$ does not. Let t be the time point where the latter inequality is not true, that is, $\sum_{j=1}^{n} \pi_j x_j > \pi_0$ for $x_j = \langle s_j \mid d_j \rangle_t$.

We thus have constructed a vector $\boldsymbol{x} \in \{0,1\}^n$ such that $A\boldsymbol{x} \leq \boldsymbol{b}$ and $\boldsymbol{\pi}^T \boldsymbol{x} > \pi_0$. However, that contradicts the definition of the valid inequality, since $\boldsymbol{x}$ is a point in the polyhedron $\mathcal{P}(A, \boldsymbol{b})$ for which the valid inequality $\boldsymbol{\pi}^T \boldsymbol{x} \leq \pi_0$ does not hold. ◀

The inference transfer lemma serves a foundational role in the approach presented in this paper, as it describes the inferred `Cumulative` constraints in terms of another well-studied object: valid inequalities of a polyhedron. On the other hand, Lemma 8 does not *prescribe* any approach for constructing valid inequalities. However, the previous subsection introduced such a procedure in the form of lifting. To see how the concepts introduced in this section – occupancy vectors, valid inequalities, and lifting – tie together, consider the following example:

▶ **Example 9.** Consider a CSP on variables $\boldsymbol{s} = (s_1, s_2, s_3, s_4)$ with a constraint

$$\texttt{Cumulative}(\boldsymbol{s}, (5,3,2,4), \boldsymbol{d}, 7) \tag{5}$$

for some vector of durations $\boldsymbol{d}$. In polyhedral view, this constraint can be written as

$$5\langle s_1 \mid d_1 \rangle + 3\langle s_2 \mid d_2 \rangle + 2\langle s_3 \mid d_3 \rangle + 4\langle s_4 \mid d_4 \rangle \leq 7.$$

To infer a new `Cumulative` constraint, we use Lemma 8 to look for valid inequalities of the following polyhedron:

$$\{(x_1, x_2, x_3, x_4) \in \{0,1\}^4 : 5x_1 + 3x_2 + 2x_3 + 4x_4 \leq 7\}. \tag{6}$$

Observe that $C = \{2,3,4\}$ is the *cover* for Equation 6 ($3 + 2 + 4 = 9 > 7$), and it corresponds to the *cover inequality* $x_2 + x_3 + x_4 \leq 2$ forbidding the simultaneous choice of those three variables.

Since this is a valid inequality, we can improve it by applying the lifting procedure; in this example, we lift the only remaining variable x_1. Consider a constraint $\alpha x_1 + x_2 + x_3 + x_4 \leq 2$ for arbitrary $\alpha \geq 0$ and observe that it holds for any solution satisfying Equation 6 with $x_1 = 0$ regardless of α. Suppose that $x_1 = 1$, and rewrite the lifted constraint and Equation 6 as follows:

$$\underbrace{3x_2 + 2x_3 + 4x_4 \leq 2}_{\text{Equation 6}} \implies \underbrace{\alpha \leq 2 - (x_2 + x_3 + x_4)}_{\text{lifted constraint for } x_1 = 1}. \tag{7}$$

Since we look for the tightest valid constraint, we can achieve this by choosing the largest α such that Equation 7 is still true for any $x_2, x_3, x_4 \in \{0,1\}$. Specifically, we achieve this by setting

$$\alpha \leftarrow 2 - \max\{x_2 + x_3 + x_4 | 3x_2 + 2x_3 + 4x_4 \leq 2, x_2, x_3, x_4 \in \{0,1\}\}, \tag{8}$$

which also happens to be the value of the lifting subproblem v^* in Equation 4 for the values of $(A, \boldsymbol{b}, C, \boldsymbol{\pi}, \pi_0)$ used in this example. To solve Equation 8, observe that $x_3 = 1, x_2 = x_4 = 0$ is a feasible solution, but setting any two variables to one is not possible under the $3x_2 + 2x_3 + 4x_4 \leq 2$ constraint. Therefore, setting $\alpha \leftarrow 2 - 1 = 1$ retains the validity of the inequality, and we obtain the lifted inequality $\boxed{x_1 + x_2 + x_3 + x_4 \leq 2.}$

Going back to the original problem (inferring a new `Cumulative` constraint), we can summarize the reasoning above as follows: the constraint in Equation 5 implies

$$\texttt{Cumulative}(\boldsymbol{x}, (1,1,1,1), \boldsymbol{d}, 2).$$

In other words, we have discovered that *at most two tasks can run in parallel* in any feasible assignment. ⌟

Before proceeding, it is important to point out that the discovery of valid inequalities, while similar to its integer programming counterpart, has a subtle difference from it: the goal in the integer programming case is usually not to merely produce new inequalities, but to *separate* a non-integer solution $\boldsymbol{x}^*$ from the linear programming relaxation by a valid inequality [17]; this, in turn, suggests starting with a cover violated by $\boldsymbol{x}^*$. In the absence of such guidance, some other strategy for enumerating covers is required. The next section describes these steps—cover generation, lifting, and constraint introduction—in more detail.

4 Inferring `Cumulative` constraints by lifting

This section introduces an approach for inferring auxiliary `Cumulative` constraints from m constraints $\texttt{Cumulative}(\boldsymbol{s}, \boldsymbol{a}_i, \boldsymbol{d}, b_i)$, $i = 1, \ldots, m$. To employ lifting in the absence of a guiding non-solution, I propose to:

1. Find a collection $\mathcal{C}$ of at most N_{cover} covers for the constraints $A\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \in \{0,1\}^n$.
2. For each cover $C \in \mathcal{C}$, lift the cover inequality $\sum_{c \in C} x_c \leq |C| - 1$ with respect to $\mathcal{P}(A, \boldsymbol{b})$ as described in Subsection 3.2.
3. Among the lifted `Cumulative` constraints, add a limited number N_{out} of them, giving preference to the most restrictive constraints.

To rank constraints by restrictiveness in Steps 1 and 3, I use the following quality metric:

▶ **Definition 10.** *Given a constraint* $\texttt{Cumulative}(\boldsymbol{s}, \boldsymbol{\pi}, \boldsymbol{d}, \pi_0)$*, its* ***capacity bound*** *is the ratio of total resource usage over the time horizon to its capacity per time unit:* $\mathcal{L}(\boldsymbol{a}, \boldsymbol{d}, b) := \frac{\sum_i d_i \pi_i}{\pi_0}$.

The relevance of this quantity, as pointed out in the early RCPSP work [34, p. 255] [18, Section 2], is that it gives a lower bound on the duration spanned by the tasks:

▶ **Lemma 11.** *For any satisfying assignment of a constraint* $\texttt{Cumulative}(\boldsymbol{s}, \boldsymbol{\pi}, \boldsymbol{d}, \pi_0)$*, let* $\text{EST} := \min_i s_i$ *and* $\text{LFT} := \max_i (s_i + d_i)$*. Then* $\text{LFT} - \text{EST} \geq \mathcal{L}(\boldsymbol{\pi}, \boldsymbol{d}, \pi_0)$.

The complete procedure is given by Algorithm 1 (cover enumeration) and Algorithm 2 (lifting and constraint selection). Aside from the main workflow—find a cover, lift it, score the resulting constraint—the implementation employs two additional optimizations. First, it discards any valid inequality $(\boldsymbol{\pi}, \pi_0)$ dominated by a model constraint, in the sense that $\boldsymbol{\pi} \leq \boldsymbol{a}_i$ and $\pi_0 \geq b_i$ for some i. Second, after lifting a cover C to a constraint with support $C' \supseteq C$, all subsets of C' of cardinality $|C|$ are marked as covers not requiring further lifting; the following example illustrates why this is beneficial.

▶ **Example 12.** Suppose that Algorithm 2 has lifted a cover $C_0 = \{1, 2\}$ to a constraint

$$\underbrace{\langle x_1 \mid d_1\rangle + \langle x_2 \mid d_2\rangle}_{\text{cover inequality}} + \underbrace{\langle x_3 \mid d_3\rangle + \cdots + \langle x_{100} \mid d_{100}\rangle}_{\text{lifted terms}} \leq 1. \tag{9}$$

In other words, lifting has discovered that no two tasks among the set $C' = \{1, 2, \ldots, 100\}$ can run in parallel, which translates to an update $Q \leftarrow Q \cup (C', 2)$.

Now, consider later iterations of the loop over covers C that visit other binary covers within C'; since any two tasks form a cover, this happens for $\binom{100}{2} - 1 = 4949$ iterations unless some task pairs were discarded before the lifting. Observe that the condition for skipping holds for *any two tasks* in C' since $(C', 2) \in Q, C' \supseteq C$, and $|C| \geq 2$; that means that none of those iterations invokes the v^* function, and skipping ≈ 5000 redundant calls to v^* is a performance gain, because (a) solving lifting subproblems is the bottleneck of Algorithm 2, and (b) lifting on C' is likely to re-discover Equation 9 instead of a new constraint; for example, it is guaranteed to do so if it lifts the variables in C' before others.

Algorithm 1 Enumeration of covers.

Input: Polyhedron $\mathcal{P}(A, \boldsymbol{b})$, durations $\boldsymbol{d}$, cap N_{cover}.
Output: A set of covers $\mathcal{C}$.

C1. **[Short covers.]** For each resource r, scan all task pairs $\{i, j\}$. If $a_{ri} + a_{rj} > b_r$, add $\{i, j\}$ as a binary cover. Otherwise, if another task k satisfies $a_{rk} > b_r - (a_{ri} + a_{rj})$, add the cover $\{i, j, k^*\}$, where k^* is a task with the longest duration among the ones satisfying $a_{rk} > b_r - (a_{ri} + a_{rj})$.

C2. **[Filter.]** Retain the N_{cover} short covers with the highest capacity bound $\mathcal{L}(\cdot, \boldsymbol{d}, \cdot)$, discarding the rest.

C3. **[Long covers.]** For each resource r and each consumption value v appearing on $\boldsymbol{a}_r$, let $B := \{i : a_{r,i} = v\}$ and let k be the smallest integer with $kv > b_r$. If $|B| \geq k$, add as covers the k longest and the k shortest tasks in B.

Algorithm 2 Lifting-based inference of auxiliary `Cumulative` constraints.

Input: Polyhedron $\mathcal{P}(A, \boldsymbol{b})$ on n variables, durations $\boldsymbol{d}$, start-time variables $\boldsymbol{s}$.
Input: Covers $\mathcal{C}$, cap N_{out}, lifting subproblem call budget N_{calls}.
Output: A set of at most N_{out} valid `Cumulative` constraints.

L1. **[Initialize.]** Set $Q \leftarrow \emptyset$ (the set of known cover supersets) and $O \leftarrow \emptyset$ (the output constraints). Set the remaining call budget $\rho \leftarrow N_{\text{calls}}$.

L2. **[Process next cover.]** Take the next cover $C \in \mathcal{C}$. If $\exists (C', k) \in Q$ with $C' \supseteq C$ and $k \leq |C|$, skip to **L2**. Otherwise, set $\pi_i \leftarrow [\![i \in C]\!]$ for $i = 1, \ldots, n$, $\pi_0 \leftarrow |C| - 1$, and $I \leftarrow \{1, \ldots, n\} \setminus C$.

L3. **[Lift next variable.]** If $I = \emptyset$ or $\rho = 0$, go to **L4**. Otherwise choose $i \leftarrow \arg\min_{i \in I} d_i$, solve the lifting subproblem $v^*(\boldsymbol{\pi}, \pi_0, C, i; A, \boldsymbol{b})$, set $\pi_i \leftarrow \pi_0 - v^*(\boldsymbol{\pi}, \pi_0, C, i; A, \boldsymbol{b})$, update $I \leftarrow I \setminus \{i\}$, $C \leftarrow C \cup \{i\}$, and $\rho \leftarrow \rho - 1$. Repeat.

L4. **[Record.]** Update $Q \leftarrow Q \cup (\{i : \pi_i = 1\}, |C_0|)$ where C_0 is the original cover. If no existing constraint r satisfies $\boldsymbol{a}_r \geq \boldsymbol{\pi}$ and $b_r \leq \pi_0$, add $\texttt{Cumulative}(\boldsymbol{s}, \boldsymbol{\pi}, \boldsymbol{d}, \pi_0)$ to O. If $\mathcal{C}$ is not exhausted, go to **L2**.

L5. **[Select.]** Return the N_{out} constraints in O with the highest value of $\mathcal{L}(\cdot, \boldsymbol{d}, \cdot)$.

5 Experimental evaluation

In this section, I evaluate the performance of the presented approach on a collection of research benchmarks. The implementation of the approach, together with the infrastructure for running the experiments and processing their results, is available in the supplementary material. The modeling part is based on the CPMpy library [12]; all lifting subproblems (Equation 4) are solved with Gurobi 12 [13]. I ran the experiments on DelftBlue [9], with each run of an instance being allocated a single core of an Intel Xeon E5-6248R 24C 3.0GHz processor and 4000 MB of RAM with a time limit of one hour.

I use the benchmark collection previously used by Sidorov et al. [32] to evaluate the UnL approach for the same two problem models (RCPSP and RCPSP/max) with `Cumulative` constraints and difference constraints of the form $x_j - x_i \geq \gamma_{ij}$ for RCPSP/max and $x_j - x_i \geq d_i$ for RCPSP, both having the latest completion time of all tasks (makespan) as the minimization objective. More specifically, I use 736 RCPSP instances from MiniZinc benchmarks [36, 25, 3, 6, 20, 19] and 349 RCPSP/max benchmarks distributed by PSPLIB [19] through C, D, UBO, and SM suites. All data files are available in the supplementary materials.

Given a minimization objective $\mathcal{O}$, each of the solvers is run with one of the two search directions. In the **primal search**,[3] the solver generates a series of problems with an extra assumption $[\![\mathcal{O} \leq o_n]\!]$ for decreasing values of o_n. In the **dual search**,[4] the solver generates a series of problems with an extra assumption $[\![\mathcal{O} \leq o_n]\!]$ for increasing values of o_n.

Each of the runs involves one of the following CP solvers: Pumpkin[5] [11] and CP-SAT [28]. The choice of the solvers is dictated by two factors: (a) both of the mentioned codes are advanced, state-of-the-art solvers performing well at the recent editions of MiniZinc Challenge, and (b) both of them support both the primal and the dual search directions. In the baseline runs, the solver is run on the input CSP as is, whereas the runs with the lifting first execute Algorithm 1 with $N_{\text{cover}} = 100$, then Algorithm 2 with $N_{\text{out}} = 5$, and proceed with the solver execution on the augmented CSP in the remaining time.

All CP solver configurations use default presolve settings: CP-SAT performs structural simplifications without inferring new cumulative constraints, and its lifting-based cut generation operates on energy overflows in the linear programming relaxation rather than on occupancy vectors. Pumpkin performs no presolve. Therefore, I do not expect overlap with the inferences produced by the presented approach.

I budget the preprocessing by limiting the number of candidate covers considered (N_{cover}) and the number of added inferred constraints (N_{out}), rather than by imposing a wall-time cutoff, since wall-time on shared HPC systems is not deterministic. All solvers were executed with the free search option.

To evaluate the solver performance when it did not conclude optimality, I measure the rate of progress of a solver towards the best-known bound with the following metrics. First, if $M(t)$ is the lowest makespan discovered at time $t \in [0, T]$, and M^* is the lowest discovered makespan for this problem, then the **primal integral** is $\int_0^T \frac{M(t)-M^*}{M(t)}$ and measures how fast the solver progresses towards good solutions [5]. Conversely, if $B(t)$ is the highest lower bound discovered at time $t \in [0, T]$, and B^* is the highest discovered lower bound for this problem, the **dual integral** is defined as $\int_0^T \frac{B^*-B(t)}{B^*}$.

[3] Also known as the linear SAT-UNSAT search.
[4] Also known as the linear UNSAT-SAT search or destructive lower bound.
[5] The version distributed in supplementary materials outputs intermediate lower bounds.

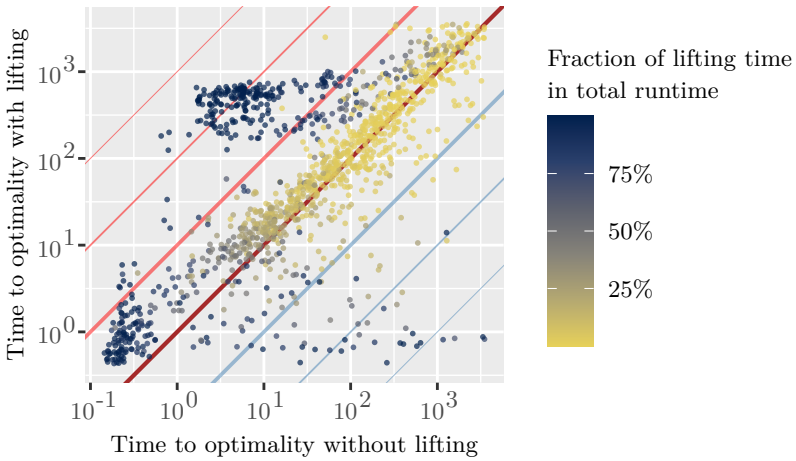


Figure 1 Impact of using lifting on time to optimality. Every point corresponds to a pair of runs on the same instances, using the same underlying solver with the same search direction. Diagonal lines are evenly spaced and correspond to a tenfold relative change between the durations.

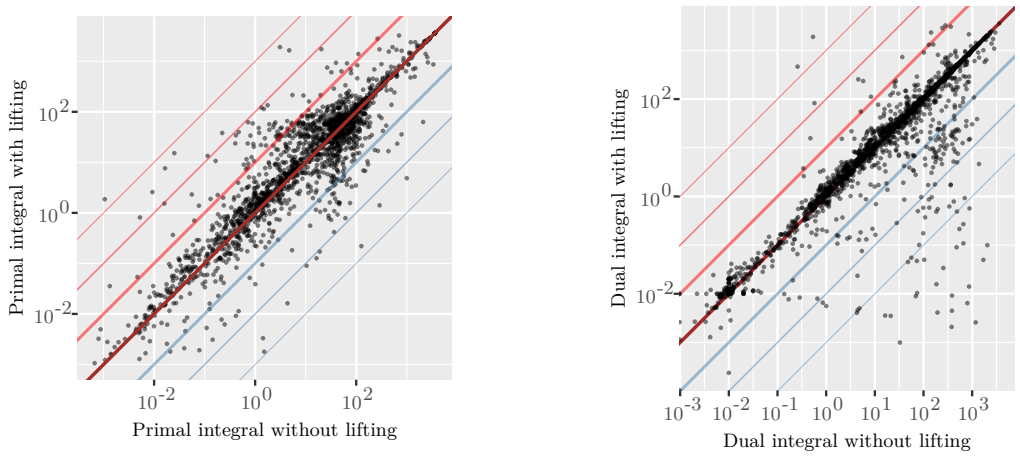
5.1 Evaluation against direct CP solver application

First, I report the evaluation of the lifting approach for the instances where both the run performing lifting and its counterpart that does not lift (while having the same settings otherwise) have proven optimality. As shown on Figure 1, lifting can cut down the solving time by large factors, sometimes from taking dozens of minutes to sub-second times. On the other hand, most of the instances with large (tenfold or more) slowdowns correspond to the cases where the lifting time dominates the solving time; more specifically, those are UBO500 and UBO1000 instances[6] for which the number of lifting subproblems exceeds 10^4. Appendix C evaluates for these families a variant of the lifting configuration with an additional termination criterion of 2×10^4 lifting subproblems, showing that it eliminates the majority of preprocessing-dominated slowdowns at the cost of occasionally weaker lifted constraints.

However, most of the instances were not solved to optimality with either approach, which means that making progress with further comparisons is more viable with primal and dual integrals as metrics. I proceed with examining this comparison on Figure 2b, which supports the thesis that lifting is a helpful preprocessing step for well-suited instances: it reduces the search effort on many instances, while typically not incurring prohibitive search overhead if it backfires. On the flip side (Figure 2a), this is not true for primal integrals: in other words, adding new constraints does not help discover good solutions as consistently, but can considerably hinder this process.

These results already show an improvement over the UnL, since Sidorov et al. mention a much larger variance for dual integrals, especially on the degradation side. For example, they report that their approach degrades the search by at least an order of magnitude on many instances, whereas in the current evaluation, this is an *exceptionally* rare event happening only in 19 pairs of runs. I investigate this in more detail in Table 1, which shows that observing the degradation by at least a factor of 2 or larger is around the bottom 5% of

[6] All instances from other collections take fewer than 2×10^4 lifting subproblems to terminate.



(a) Primal integrals.

(b) Dual integrals.

Figure 2 Comparison of integrals between the baseline and the lifting approach.

Table 1 Percentiles of the observed distribution of the ratio of dual integrals between baseline runs and the corresponding runs with lifting across problem types; values larger than one indicate an advantage for lifting.

Problem type	2.5%	5%	25%	50%	75%	95%	97.5%
RCPSP	0.33	0.48	0.85	1.00	1.05	10.7	47.4
RCPSP/max	0.41	0.50	0.77	0.97	1.41	51.0	292

observed integral ratios, whereas the top 5% of the runs correspond to 10× improvements for RCPSP benchmarks and to 50× improvements for RCPSP/max benchmarks.

I also observe that in many high-performing runs of the lifting approach, the underlying constraints – or, at least, the one yielding the highest capacity bound as introduced in Definition 10 – commonly happened to be a `Disjunctive` constraint, that is, to have the right-hand side equal to one. In a similar vein, I investigate how the influence of lifting on the search depends on the constraint with the highest capacity bound, which is summarized in Figure 3. Indeed, the lifting runs that ended up on a `Disjunctive` constraint as the best one are responsible for the biggest relative gains; however, the runs that landed on a `Cumulative` constraint still routinely produce large relative improvements, even if more modest. That said, discovering general `Cumulative` constraints is only helpful when the capacity of the discovered constraint is not large; conversely, in the runs where the best constraint inferred through lifting had a capacity larger than eight units, the CP solvers have not been able to convert it into a significantly faster search.

I present the runtime distribution of the preprocessing phase of the lifting approach (that is, Algorithm 1 and Algorithm 2 but not the CP solver) in Figure 4. As can be seen on the histogram, under the budget imposed on lifting in this evaluation, all the lifting-specific work takes a few seconds in the majority of instances; on the other hand, the runs where lifting took more than a minute are covered exclusively by two collections with more than 500 tasks per instance, that is, UBO500 and UBO1000.

Last, I report the updated state-of-the-art bounds for these benchmarks discovered with the lifting approach in Appendix A, together with the criterion I used to choose which bounds

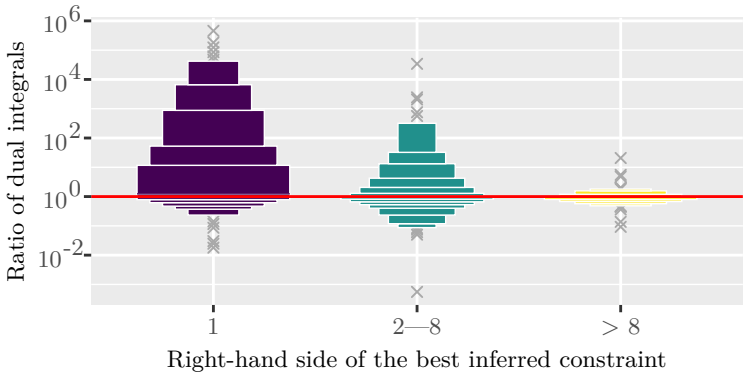


Figure 3 Ratios of dual integrals between baseline runs and the corresponding runs with lifting across the right-hand side values of the best inferred constraint; values larger than one indicate an advantage for lifting.

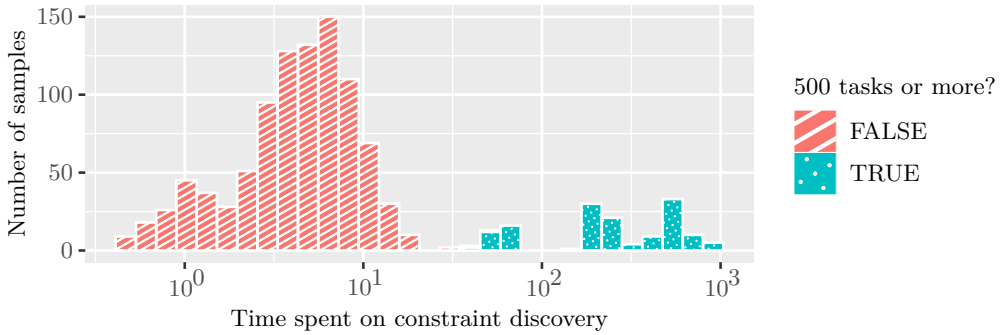


Figure 4 Distribution of time required to run the inference of `Cumulative` constraints.

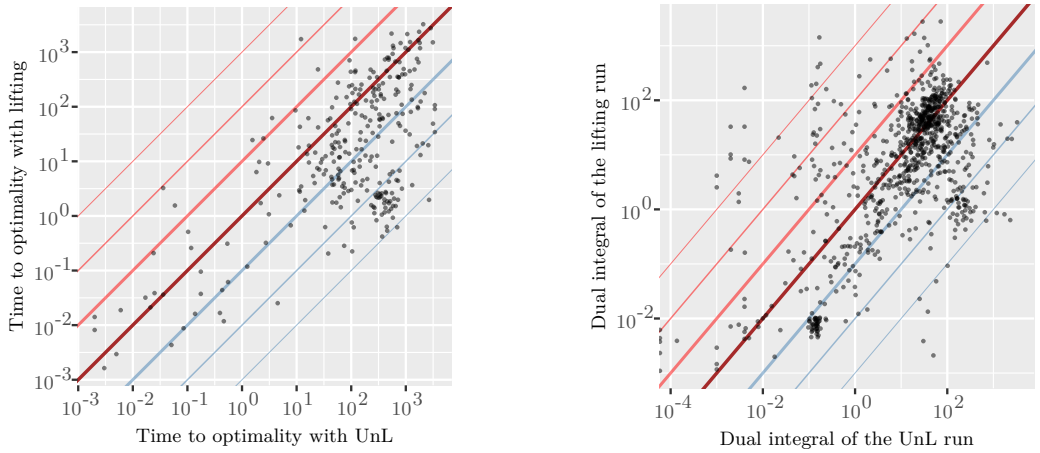
to report. While many of the new bounds have been discovered during search, some are computed by taking the largest capacity bound among lifted constraints, thereby making them verifiable by inspecting the constraint (rather than retracing the reasoning made by a solver).[7] While many such search-less bounds are encoded by `Disjunctive` constraints (and therefore could, in principle, have also been proven by UnL without any search), there is a bound for instance #6 from the UBO1000 collection that improves upon the previously best known bound and is produced by a `Cumulative` with the capacity of three units. I also note that no less than twelve of the instances in Pack and Pack-d collections [6, 20] can be closed immediately by using one of the lifted cumulative constraints, with the capacities varying between one and three.

5.2 Evaluation against UnL

I now proceed with the direct comparison with UnL; to this end, I run the following two solver configurations on each of the benchmark instances:

- UnL with dual search in the same configuration as in Sidorov et al.
- Pumpkin with dual search, prefaced with lifting in the same configuration as in the

[7] See also the supplementary material for solver-independent verification of the search-less bounds.



(a) Time to optimality for the instances solved to optimality with both approaches.

(b) Dual integrals for all evaluated instances.

Figure 5 Comparison between the lifting approach and UnL.

previous subsection, *but only considering covers with two elements*. This additional restriction is introduced to ensure that the lifting does not gain any advantage from discovering non-`Disjunctive` constraints.

As can be seen from Figure 5, the proposed approach outperforms UnL with respect to time to optimality on the majority of the applicable instances (Figure 5a) and has varying performance with respect to the dual integral (Figure 5b). This is explained by the fact that the lifting approach explicitly introduces `Disjunctive` constraints into the model, which are then used in specialized—and highly optimized—propagation algorithms. In contrast, UnL maintains all the facts around the disjointness of task pairs in a single graph and applies pruning for some of the cliques of that graph.

These plots also suggest that the lifting approach does not fully subsume UnL, as for some instances, lifting yields a notably worse performance. Interestingly, while many instances favorable for UnL are adequately explained by reasons unrelated to the methodology of this paper (such as the implementation of the preprocessing stage or discrepancies between the underlying solver versions), this is not true in all cases. For example, instance #40 from the UBO50 collection is solved to optimality for ≈ 30 seconds with UnL and for ≈ 8 minutes with lifting, with UnL also doing $90\times$ fewer constraint propagations until optimality. The reason for this is that the lifting approach chooses N_{out} constraints with high capacity bounds (which in this case is the sum of durations) that also happen to share many variables, which translates into an insufficient extraction of the disjointness information; this reasoning is made more specific in Appendix B.

6 Related work

The need to reason simultaneously about multiple resources has been recognized across optimization communities throughout the history of earlier developments. For instance, early exact approaches for RCPSP consistently employed global lower-bounding techniques that used the entire problem instance structure, such as the lower bounds mentioned by Klein and Scholl [18] based on solving relaxations to node packing (LB4) and parallel machine

scheduling (LB5) problems. The presented approach can be seen as a generalization of these techniques in the following sense: Algorithm 2 with an appropriate choice of the input covers and an ordering of lifted variables can yield the same bounds as LB4 or LB5 implementations by letting a CP solver propagate the lifted `Cumulative` constraints. However, unlike using LB4 or LB5 directly, the new `Cumulative` remains known to a CP solver as one of the constraints rather than as an exogenous relaxation step, with any propagation algorithms for `Cumulative` applicable to it.

Within the CP community, the limitations of the single-resource viewpoint were also acknowledged for a long time. Beldiceanu and Carlsson [4] introduced the `Cumulatives` constraint to capture multiple resources in a single constraint, which was later improved by faster filtering algorithms [22, 23]. While these works highlighted the need for multi-resource interaction, their focus was not on discovering new inference strategies but on running existing strategies faster, by replacing what they call the *ping-pong* effect, where propagation for one `Cumulative` triggers a propagation for another, with a single pass accounting for all `Cumulative` constraints. Conversely, the lifting approach can not only draw conclusions from several `Cumulative` constraints at once but can also draw conclusions unreachable (without an exponential blowup) by reasoning over them in isolation.

Other efforts to mitigate the weaknesses of the single-resource viewpoint have focused on extracting stronger inferences from individual constraints by deriving non-dominated implied constraints. Carlier and Néron [7] investigated the systematic derivation of such constraints for a single `Cumulative` by searching for one-to-one mappings from original resource consumptions to a reduced set of demands that preserve infeasibility while maximizing pruning. This logic was further refined by Baptiste and Bonifas [1] by examining the dual linear program of a preemptive relaxation of `Cumulative` and introducing a mechanism for assigning a higher consumption level to a fixed number of tasks within a demand group. While these techniques strengthen the reasoning of a single resource, they remain limited to intra-resource analysis. This work instead uses lifting as a tool for *inter*-resource reasoning; rather than refining a single `Cumulative` in isolation, the presented work derives global inequalities that capture the collective contention of tasks across the entire problem structure.

The earlier work by Sidorov et al. [33] implements a whole-problem reasoning scheme without merging constraints, opting instead to discover and communicate disjointness between tasks across resources. While UnL can theoretically emulate the node-packing bound (LB4) of Klein and Scholl, it lacks the expressive power to capture variable non-unit demands (LB5). Also, it relies on managing auxiliary variables during search, which can be both a strength (by discovering a disjointness conditional to a search state) and a hindrance (if there is no useful disjointness structure). Conversely, the presented work compiles these interactions into conventional `Cumulative` constraints before initiating any search by a CP solver.

The core mechanism that supports the contributions of this paper—lifting—originated in the integer programming community as a method for tightening cutting planes for knapsack constraints. First introduced by Padberg [27], the technique iteratively strengthens an inequality by introducing variables not yet in the support. Later refinements, such as the ones proposed by Zemel [40], demonstrated that all sequential lifting coefficients could be computed efficiently via dynamic programming. Currently, lifting is a standard part of the modern MILP solver practice, as exemplified by the SCIP optimization suite [15], which extensively employs lifting procedures within its knapsack constraint handler.

In the context of scheduling, polyhedral methods have been originally applied to single-machine problems [29, 10]. For the general RCPSP, early polyhedral approaches [26] focused on conflict variables (indicating a precedence within a pair of tasks) rather than start

times, limiting their direct applicability in standard CP models. More recently, approaches combining CP propagators with LP relaxations have been explored, with CP-SAT [28], one of the state-of-the-art CP solvers, integrating scheduling cuts as special separation procedures for tightening LP relaxations. While scheduling cuts are constructed with a procedure similar to the one proposed here (choose a cover and lift it), those cuts are derived from *energy*[8] overflows; on the other hand, lifting as presented in this paper operates directly on the *demands* via occupancy vectors, allowing us to view `Cumulative` constraints as linear inequalities and apply standard lifting machinery directly to the start-time representation.

Collectively, these works support the thesis that the single-resource viewpoint is insufficient for complex scheduling benchmarks. While integer programming and specialized branch-and-bound approaches have long exploited whole-problem structures, CP has historically struggled to integrate these insights without sacrificing the modularity of propagators. By establishing a formal link between `Cumulative` constraints and polyhedral geometry, this work offers a pragmatic path to bring these powerful global inferences into CP solvers.

7 Conclusions

This paper presents a novel approach for aggregating `Cumulative` constraints by (i) reformulating them as linear constraints in the occupancy-vector representation, (ii) discovering sets of tasks that cannot be run jointly, and (iii) lifting the linear constraint blocking this group of tasks. The experimental evaluation shows that this approach can, at its best, be substantially helpful—or, at its worst, not as obstructive as UnL—for proving objective bounds, although those advantages do not translate as well to discovering new high-quality solutions.

One possible direction for future work is a tighter coupling with the search loop, for instance, by adding `Cumulative` constraints not *before* the search but *during* the search. The core technical problem here is that a `Cumulative` lifted from a conflicting assignment does not necessarily propagate after backtracking, as observed in earlier work on linear constraint learning in CP [24]. However, exploiting this procedure in a more heuristic way (e.g., during restarts) could be helpful to leverage the information learned during search.

Last, the running assumption in this work is that both durations and resource consumptions are constant. While this is valid for RCPSP and RCPSP/max, this is not true for several other scheduling problem formulations studied previously, most notably *multi-mode* scheduling problems [14]. However, extending this approach to multi-mode RCPSP—where task durations and demands are variable—will require handling the non-linearities resulting from the products of occupancy vectors and resource consumptions.

References

1 Philippe Baptiste and Nicolas Bonifas. Redundant cumulative constraints to compute preemptive bounds. *Discrete applied mathematics*, 234:168–177, 2018. `doi:10.1016/j.dam.2017.05.001`.

2 Philippe Baptiste, Claude Le Pape, and Wim Nuijten. Satisfiability tests and time-bound adjustments for cumulative scheduling problems. *Annals of operations research*, 92:305–333, 1999. `doi:10.1023/a:1018995000688`.

[8] The product of duration and resource consumption.

3 Philippe Baptiste and Claude Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints: an international journal*, 5(1/2):119–139, 2000. doi:10.1023/a:1009822502231.

4 Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In *Principles and practice of constraint programming - CP 2002*, volume 2470 of *Lecture Notes in Computer Science*, pages 63–79, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-46135-3_5.

5 Timo Berthold. Measuring the impact of primal heuristics. *Operations research letters*, 41(6):611–614, 2013. doi:10.1016/j.orl.2013.08.007.

6 Jacques Carlier and Emmanuel Néron. On linear lower bounds for the resource constrained project scheduling problem. *European journal of operational research*, 149(2):314–324, 2003. doi:10.1016/s0377-2217(02)00763-4.

7 Jacques Carlier and Emmanuel Néron. Computing redundant resources for the resource constrained project scheduling problem. *European journal of operational research*, 176(3):1452–1463, 2007. doi:10.1016/j.ejor.2005.09.034.

8 Edward W Davis. Resource allocation in project network models — a survey. *Journal of Industrial Engineering*, 17(4):177–188, 1966. ISSN: 1045-0165.

9 Delft High Performance Computing Centre. DelftBlue, 2024.

10 Martin E Dyer and Laurence A Wolsey. Formulating the single machine sequencing problem with release dates as a mixed integer program. *Discrete Applied Mathematics*, 26(2-3):255–270, 1990. doi:10.1016/0166-218x(90)90104-k.

11 Maarten Flippo, Konstantin Sidorov, Imko Marijnissen, Jeff Smits, and Emir Demirović. A multi-stage proof logging framework to certify the correctness of CP solvers. In *30th international conference on principles and practice of constraint programming (CP 2024)*, pages 11:1–11:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CP.2024.11.

12 Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, CPpy as python-embedded example. In *Proceedings of the 18th workshop on constraint modelling and reformulation at CP (ModRef 2019)*, volume 19, 2019.

13 Gurobi Optimization, LLC. *Gurobi optimizer reference manual*, 2024.

14 Sönke Hartmann and Andreas Drexl. Project scheduling with multiple modes: A comparison of exact algorithms. *Networks. An International Journal*, 32(4):283–297, 1998. doi:10.1002/(sici)1097-0037(199812)32:4<283::aid-net5>3.0.co;2-i.

15 Christopher Hojny, Mathieu Besançon, Ksenia Bestuzheva, Sander Borst, João Dionísio, Johannes Ehls, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Adrian Göß, Alexander Hoen, Jacob von Holly-Ponientzietz, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Marco Lübbecke, Stephen J Maher, Paul Matti Meinhold, Gioni Mexi, Til Mohr, Erik Mühmer, Krunal Kishor Patel, Marc E Pfetsch, Sebastian Pokutta, Chantal Reinartz Groba, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Matthias Walter, Dieter Weninger, and Liding Xu. The SCIP optimization suite 10.0. *arXiv [math.OC]*, 2025. doi:10.48550/arXiv.2511.18580.

16 Christopher Hojny, Tristan Gally, Oliver Habeck, Hendrik Lüthen, Frederic Matter, Marc E Pfetsch, and Andreas Schmitt. Knapsack polytopes: a survey. *Annals of operations research*, 292(1):469–517, 2020. doi:10.1007/s10479-019-03380-2.

17 Konstantinos Kaparis and Adam N Letchford. Separation algorithms for 0-1 knapsack polytopes. *Mathematical programming*, 124(1-2):69–91, 2010. doi:10.1007/s10107-010-0359-5.

18 Robert Klein and Armin Scholl. Computing lower bounds by destructive improvement: An application to resource-constrained project scheduling. *European journal of operational research*, 112(2):322–346, 1999. doi:10.1016/s0377-2217(97)00442-6.

19 Rainer Kolisch and Arno Sprecher. PSPLIB - a project scheduling problem library: OR Software - ORSEP Operations Research Software Exchange Program. *European Journal of Operational Research*, 96(1):205–216, 1997. doi:10.1016/S0377-2217(96)00170-1.

612 **20** Oumar Koné, Christian Artigues, Pierre Lopez, and Marcel Mongeau. Event-based MILP
613 models for resource-constrained project scheduling problems. *Computers & operations research*,
614 38(1):3–13, 2011. doi:10.1016/j.cor.2009.12.011.
615 **21** Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP optimizer
616 for scheduling. *Constraints: An International Journal*, 23(2):210–250, 2018. doi:10.1007/
617 s10601-018-9281-x.
618 **22** Arnaud Letort, Mats Carlsson, and Nicolas Beldiceanu. A synchronized sweep algorithm
619 for the k-dimensional cumulative constraint. In *Integration of AI and OR techniques in*
620 *constraint programming for combinatorial optimization problems*, volume 7874 of *Lecture Notes*
621 *in Computer Science*, pages 144–159, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
622 doi:10.1007/978-3-642-38171-3_10.
623 **23** Arnaud Letort, Mats Carlsson, and Nicolas Beldiceanu. Synchronized sweep algorithms for
624 scalable scheduling constraints. *Constraints: An International Journal*, 20(2):183–234, 2015.
625 doi:10.1007/s10601-014-9172-8.
626 **24** Robert Nieuwenhuis. The IntSat method for integer linear programming. In *Principles and*
627 *practice of constraint programming*, volume 8656 of *Lecture notes in computer science*, pages 574–
628 589, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-10428-7_42.
629 **25** Ramón Alvarez-Valdés Olaguíbel and José Manuel Tamarit Goerlich. Heuristic algorithms for
630 resource-constrained project scheduling: A review and an empirical analysis. In *Advances in*
631 *project scheduling*, volume 9 of *Studies in Production and Engineering Economics*, chapter 5,
632 pages 113–134. Elsevier, North Holland, 1989. doi:10.1016/b978-0-444-87358-3.50009-2.
633 **26** Ramón Alvarez-Valdés Olaguíbel and Josémanuel Tamarit Goerlich. The project scheduling
634 polyhedron: Dimension, facets and lifting theorems. *European Journal of Operational Research*,
635 67(2):204–220, 1993. doi:10.1016/0377-2217(93)90062-r.
636 **27** Manfred W Padberg. Technical note–a note on zero-one programming. *Operations research*,
637 23(4):833–837, 1975. doi:10.1287/opre.23.4.833.
638 **28** Laurent Perron and Frédéric Didier. CP-SAT. URL: https://developers.google.com/
639 optimization/cp/cp_solver/.
640 **29** Maurice Queyranne. Structure of a simple scheduling polyhedron. *Mathematical Programming*,
641 58(1-3):263–285, 1993. doi:10.1007/bf01581271.
642 **30** Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of constraint programming*.
643 Foundations of artificial intelligence. Elsevier Science, London, England, 2006. doi:10.1016/
644 s1574-6526(06)x8001-x.
645 **31** Andreas Schutt, Thibaut Feydy, Peter J Stuckey, and Mark G Wallace. Why cumulative
646 decomposition is not as bad as it sounds. In *Principles and practice of constraint programming*
647 *- CP 2009*, Lecture Notes in Computer Science, pages 746–761, Berlin, Heidelberg, 2009.
648 Springer Berlin Heidelberg. doi:10.1007/978-3-642-04244-7_58.
649 **32** Konstantin Sidorov, Imko Marijnissen, and Emir Demirović. Data collection for "Unite and
650 lead: Finding disjunctive cliques for scheduling problems", June 2025. doi:10.5281/zenodo.
651 15624416.
652 **33** Konstantin Sidorov, Imko Marijnissen, and Emir Demirović. Unite and lead: Finding disjunct-
653 ive cliques for scheduling problems. In Maria Garcia de la Banda, editor, *31st international*
654 *conference on principles and practice of constraint programming (CP 2025)*, volume 340 of *Leib-*
655 *niz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:24, Dagstuhl, Germany,
656 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2025.35.
657 **34** Joel P Stinson, Edward W Davis, and Basheer M Khumawala. Multiple resource–constrained
658 scheduling using branch and bound. *AIIE Transactions*, 10(3):252–259, 1978. doi:10.1080/
659 05695557808975212.
660 **35** Peter J Stuckey. RCPSP. https://people.eng.unimelb.edu.au/pstuckey/rcpsp/. Accessed:
661 2025-3-29.
662 **36** Peter J Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The MiniZinc
663 Challenge 2008–2013. *AI magazine*, 35(2):55–60, 2014. doi:10.1609/aimag.v35i2.2539.

37 Petr Vilím. $O(n \log n)$ filtering algorithms for unary resource constraint. In *Integration of AI and OR techniques in constraint programming for combinatorial optimization problems*, volume 3011 of *Lecture Notes in Computer Science*, pages 335–347, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-24664-0_23.

38 Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In *Integration of AI and OR techniques in constraint programming for combinatorial optimization problems*, volume 6697 of *Lecture Notes in Computer Science*, pages 230–245, Berlin, Heidelberg, 2011. Springer-Verlag. doi:10.1007/978-3-642-21311-3_22.

39 Petr Vilím, Philippe Laborie, and Paul Shaw. Failure-directed search for constraint-based scheduling. In *Integration of AI and OR techniques in constraint programming*, volume 9075 of *Lecture Notes in Computer Science*, pages 437–453, Barcelona, Spain, 2015. Springer Cham. doi:10.1007/978-3-319-18008-3_30.

40 Eitan Zemel. Easily computable facets of the knapsack polytope. *Mathematics of Operations Research*, 14(4):760–764, 1989. doi:10.1287/moor.14.4.760.

A Novel bounds

I report the bounds discovered by the lifting approach if they are *both* better than the previously reported bounds and are not directly reproducible without lifting. More precisely, I report bounds that are simultaneously (a) tighter than the bounds reported in the previous sources known to us that used the same benchmarks [19, 39, 38, 35, 33], and (b) either tighter than any bound derived without preprocessing or matches it but was derived at least ten times faster than with any other approach.

Novel upper bounds (makespans) are reported in Table 2, and novel lower bounds are reported in Table 3; the same data is available as supplementary materials. All bounds are reported for RCPSP/max benchmarks; all reported durations are in `MM:SS` format. To indicate the remaining optimality gap, I also state the best known lower bound in Table 2 and the best known makespan in Table 3; in either case, that bound is the tightest among the previously reported values and the values discovered without preprocessing. Additionally, Table 4 reports the lower bounds that can be verified by an appropriate lifted constraint.

As noted in the main text, using `Disjunctive` constraints, when possible, typically yields the largest gains, compared to non-unit `Cumulative` constraints. A natural hypothesis that can be proposed from this is that the new results should be credited to the discovery of a non-trivial collection of `Disjunctive` constraints, rather than to the lifting approach for deriving general `Cumulative` constraints. To address this, I rerun the reported instances with the lifting approach in the same configurations as in Subsection 5.1, but with a restriction on using only binary covers (as in Subsection 5.2), thereby ensuring that the lifting approach only adds `Disjunctive` constraints. The instances where this `Disjunctive`-restricted lifting has failed to discover the same objective values and lower bounds are highlighted in Table 2 and Table 3 respectively. As can be seen, in approximately half of the record-setting instances (eleven out of 23), it was essential to add `Cumulative` constraints with non-unit capacity to derive novel lower bounds; in a similar vein, only one of the five new best solutions has been discovered without adding non-unit `Cumulative` constraints.

B Disjointness structure of instance #40 from UBO50

As mentioned in the main text, UnL outperforms the lifting approach on this instance; this appendix provides a more specific explanation for this. Figure 6 demonstrates the complement of the disjointness graph of that instance, with vertices corresponding to each of

Table 2 Novel upper bounds derived with lifting. Highlighted rows report solutions that were not reproduced using only `Disjunctive` constraints.

Collection	#	Ref. objective	New objective	Time	Best bound
C	63	366	363	43:50	347
C	67	350	349	23:56	346
UBO100	8	385	383	41:56	376
UBO100	32	434	432	26:59	414
UBO200	4	893	838	29:25	605

the fifty tasks that are scheduled in this instance, and edges connect pairs $\{u, v\}$ of vertices that can be executed in parallel (that is, $a_{i,u} + a_{i,v} \leq b_i$ for all i). Vertices are grouped as follows:

Universal These are the blue vertices with solid fill that correspond to all tasks shared between **all** constraints added by lifting.

Occasional These are the orange vertices with hatching that correspond to tasks that are mentioned in **some but not all** constraints added by lifting.

Void These are the green vertices with dotted fill that correspond to tasks that are mentioned in **none** of the lifted constraints.

The important insight in the structure of this graph is that it is (a) very sparse, with a random pair of tasks having 87% probability of being disjoint, and (b) the void vertices are loosely constrained by the rest of the graph; to justify the latter, I observe that 235 out of 262 maximal independent sets of this graph involve a void vertex. This suggests that UnL is able to exploit all of the graph structure, which opens the inferences about the tasks corresponding to the void vertices that are not available to the CP solver after the lifting procedure concludes.

On the other hand, this shows the insufficiency of the selection step based solely on a quality metric of the `Cumulative` constraint. In this example, the lifting procedure chooses many independent sets (equivalently, `Disjunctive` constraints) that contain all the universal vertices and have a large total duration of tasks. Taken on a per-constraint basis, this is a good decision, as this set covers 40% of the problem variables; unfortunately, none of the void vertices can be added to this set. I believe that resolving the issue demonstrated by this instance would require a more elaborate selection procedure that rewards the *overall* coverage of the space of known conflicts, rather than the quality of each of the added constraints in isolation.

C Effect of a call budget on large RCPSP/max instances

The experimental evaluation in Section 5 uses two budget parameters to control preprocessing effort: the number of candidate covers $N_{\text{cover}} = 100$ and the number of added constraints $N_{\text{out}} = 5$. As noted in Section 5, instances from the UBO500 and UBO1000 collections generate significantly more lifting subproblems than other families under this configuration, with some instances exceeding 10^5 subproblems. This appendix evaluates the effect of an additional termination criterion that caps the total number of lifting subproblems at 2×10^4, applied to all UBO500 and UBO1000 instances with all other parameters unchanged. From now on, the configuration with this termination criterion is referred to as *budgeted*

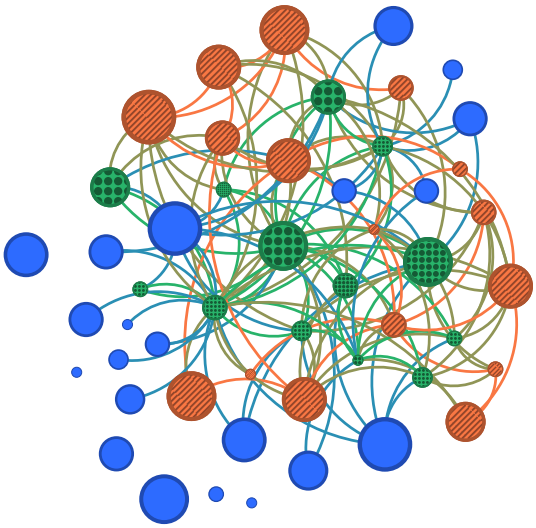
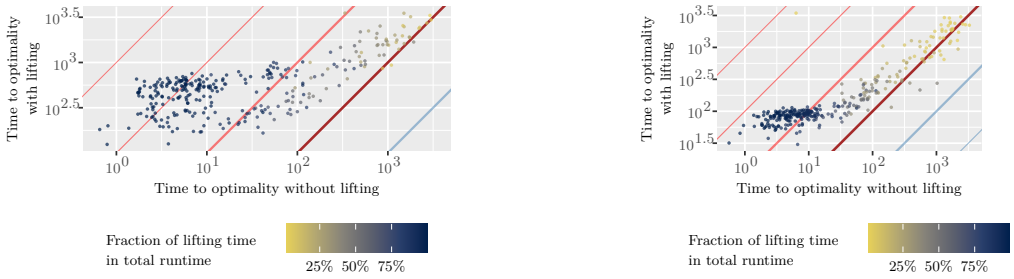


Figure 6 The complement of the disjointness graph of instance #40 from UBO50; vertex color encodes the number of lifted constraints which mention the corresponding task, and the area occupied by a vertex is proportional to the task duration.



(a) Comparison with unbudgeted configuration.

(b) Comparison with budgeted configuration.

Figure 7 Impact of using various lifting configurations on time to optimality. Every point corresponds to a pair of runs on the same instances from UBO500 or UBO1000, using the same underlying solver with the same search direction. Diagonal lines are evenly spaced and correspond to a tenfold relative change between the durations.

configuration, and the lifting configuration used in the main text is referred to as *unbudgeted* configuration.

First, I report the comparison of the time to optimality with baseline, budgeted, and unbudgeted configurations. As can be seen from Figure 7, budgeting reduces the pathological behavior pointed out in a similar comparison in the main text, as the instances where lifting dominates the runtime now take less time to terminate (because the lifting procedure takes less time). As for the rest of the instances, their performance profile has not substantially changed; the reason for that is that the most advantageous instances for lifting are the ones that were not solved to optimality, which are not reported in this comparison.

Budgeting better controls the preprocessing *overhead* on these instances; however, it also does not fully overturn the *gains* obtained from lifting. Table 5 reports the quantiles of the dual integral ratios; as can be seen, the higher end of the distribution became less pronounced, but the rest of the quantiles (until 95%) have not meaningfully changed.

Table 6 reports the instances from Table 3 where the budgeted and unbounded configurations produce different bounds after accounting for the search-less bounds from Table 4. Of the 25 novel lower bounds reported in the main text, only three are affected: the budgeted configuration discovers a search bound of 1 260 for instance #4 of UBO500, exceeding the search-less bound of 1 259 that was the best result under the unbudgeted configuration, while the bounds for UBO500 #6 and UBO1000 #68 regress. The remaining bounds are unaffected, confirming that the call budget has a negligible impact on the reported results.

One search-less lower bound (#68 from UBO1000) from Table 4 regresses under the budgeted configuration from 2 536 to 2 519, confirming that the call budget occasionally terminates lifting before a stronger constraint is discovered. This also impacts the eventual search performance: the budgeted configuration terminates at the bound 2 537, falling short of the 2 550 lower bound discovered by the configuration from the main text.

Together, these results substantiate the claim in Subsection 5.1 that a call budget eliminates the majority of preprocessing-dominated slowdowns at the cost of occasionally weaker lifted constraints.

Table 3 Novel lower bounds derived with lifting. Highlighted rows report bounds that were not reproduced using only `Disjunctive` constraints.

Collection	#	Ref. bound	New bound	Time	Best objective
C	63	343	348	43:08	363
C	66	340	347	49:16	368
UBO100	70	375	387	43:43	408
UBO200	2	682	731	46:25	813
UBO200	3	482	542	30:47	906
UBO200	5	499	559	57:28	767
UBO200	6	538	569	38:29	765
UBO200	8	505	583	48:53	911
UBO200	32	753	794	51:45	863
UBO200	33	784	785	56:14	834
UBO200	34	595	673	35:16	774
UBO200	65	728	747	35:22	814
UBO200	70	724	823	57:43	877
UBO500	3	1159	1260	43:30	1808
UBO500	6	1202	1350	01:05	2035
UBO500	8	1175	1208	00:56	2212
UBO500	38	1258	1398	01:12	2575
UBO1000	2	2321	2518	32:11	5479
UBO1000	6	2311	2329	04:12	3508
UBO1000	9	2274	2414	03:55	4231
UBO1000	10	2367	2441	02:30	3702
UBO1000	35	2382	2563	03:09	4737
UBO1000	68	2478	2550	03:12	4688

Table 4 Novel *search-less* lower bounds derived with lifting. The bounds marked with an asterisk have not been improved by a CP solver.

Collection	#	Ref. bound	New bound	Objective	Capacity
UBO200	3	482	531	906	1
UBO200	4	514	583*	838	1
UBO200	5	499	533	767	1
UBO200	6	538	558	765	1
UBO200	8	505	559	911	1
UBO500	3	1159	1260*	1808	1
UBO500	4	1230	1259	2774	1
UBO500	6	1202	1345	2035	1
UBO500	8	1175	1328*	2212	1
UBO500	38	1258	1448*	2575	1
UBO1000	2	2321	2515	5479	1
UBO1000	6	2311	2315	3508	3
UBO1000	9	2274	2418*	4231	1
UBO1000	10	2367	2482*	3702	1
UBO1000	35	2382	2568*	4737	1
UBO1000	68	2478	2536	4688	1

Table 5 Percentiles of the observed distribution of the ratio of dual integrals between baseline runs and the corresponding runs with lifting; values larger than one indicate an advantage for lifting.

Lifting configuration	2.5%	5%	25%	50%	75%	95%	97.5%
Unbudgeted	0.40	0.51	0.80	0.98	1.46	33.2	117
Budgeted	0.38	0.54	0.84	0.98	1.20	30.0	79.3

Table 6 Changes of novel lower bounds discovered by budgeted runs in comparison with unbudgeted runs.

Collection	#	Unbudgeted	Budgeted	Change direction
UBO500	4	1259	1260	↗
UBO500	6	1350	1316	↘
UBO1000	68	2550	2537	↘