A MinCumulative Resource Constraint

Yanick Ouellet and Claude-Guy Quimper

Université Laval, Québec, Canada
yanick.ouellet.2@ulaval.ca, claude-guy.quimper@ift.ulaval.ca

Abstract. The CUMULATIVE constraint is the key to the success of Constraint Programming in solving scheduling problems with cumulative resources. It limits the maximum amount of a resource consumed by the tasks at any time point. However, there are few global constraints that ensure that a minimum amount of a resource is consumed at any time point. We introduce such a constraint, the MINCUMULATIVE. We show that filtering the constraint is NP-Hard and propose a checker and a filtering algorithm based on the fully elastic relaxation used for the CUMULATIVE constraint. We also show how to model MINCUMULATIVE using the SOFTCUMULATIVE constraint. We present experiments comparing the different methods to solve MINCUMULATIVE using Constraint Programming.

1 Introduction

Businesses and organizations often face scheduling problems where they must schedule tasks while satisfying various resources constraints. In the Constraint Programming community, with the CUMULATIVE constraint [1], significant work [3,16,17] has been devoted to scheduling problems where the resource usage of the tasks must not exceed the capacity of the resource. The reverse case, where a resource must have a minimum usage did not receive as much attention. However, many businesses and organizations need to solve scheduling problems where they need to ensure that a sufficient number of employees are working at any given time.

We introduce MINCUMULATIVE, a new global constraint that enforces that a minimum amount of the resource is used at any time. We show that applying domain or bounds consistency for this new constraint is NP-Hard. We propose a relaxed rule, the `UnderloadCheck`, to detect failures and a filtering algorithm related to this rule. We also show how to model the MINCUMULATIVE using the SOFTCUMULATIVE constraint, a soft version of the CUMULATIVE. This allows us to use the strong energetic reasoning rules of the CUMULATIVE.

We present relevant background in Section 2 and introduce the MINCUMULATIVE in Section 3. We present the `UnderloadCheck` rule and checker algorithm in Section 4 while Section 5 introduces the filtering algorithm. We show how to model the MINCUMULATIVE constraint using the SOFTCUMULATIVE constraint in Section 6. We compare the difference in strength between the different approaches in Section 7. Finally, experimental results are shown in Section 8 and Section 9 concludes the work.

2 Constraint Scheduling Background

A scheduling problem consists in scheduling a set $\mathcal{I}$ of n tasks over the time points $\mathcal{T} = 0..\texttt{hor} - 1$. Each task $i \in \mathcal{I}$ needs to execute *without preemption* for p_i units of *processing time* between its *earliest starting time* est_i and its *latest completion time* lct_i and consumes h_i units of a renewable resource, which are called height. A task i can be described using the tuple $\langle i, \text{est}_i, \text{lct}_i, p_i, h_i \rangle$. One can compute the *latest starting time* $\text{lst}_i = \text{lct}_i - p_i$ and the *earliest completion time* $\text{ect}_i = \text{est}_i + p_i$ of task i. We say that task i has a *compulsory part* in the interval $[\text{lst}_i, \text{ect}_i)$ if $\text{lst}_i < \text{ect}_i$. A task is necessarily executing during its compulsory part, regardless of its starting time.

One can model a scheduling problem using a starting time variable S_i for each task i with domain $\text{dom}(S_i) = [\text{est}_i, \text{lst}_i]$. Additional constraints can be added depending on the particularity of the problem (e.g., constraints limiting the amount of resource used, predecessor constraints, etc.). We say that task i is *fixed* if there is only one value in $\text{dom}(S_i)$. Thus we have $S_i = \text{est}_i = \text{lst}_i$.

Significant efforts have been made in the constraint programming community to efficiently solve scheduling problems for which the capacity of the resource is limited. The CUMULATIVE constraint [1] enforces that, at any time t, the sum of the heights of the tasks in execution does not exceed the capacity C of the resource. Deciding whether the CUMULATIVE constraint admits a solution is strongly NP-Complete [1]. Hence, checker and filtering algorithms for this constraint cannot apply domain or bounds consistency and must instead rely on rules to partially detect failures or partially filter the domains. Many such rules have been developed over the years, including the Overload Check [9,27], the Time Tabling [3], the Edge Finding [14,17,26] and the Energetic Reasoning [2,5,16,19,25]. By using lazy clause generations [10,18] with the CUMULATIVE constraint, Schutt et al. [23,24] closed many instances of hard scheduling problems.

Baptiste et al. [2] introduced a relaxation used in many rules for the CUMULATIVE constraint: the fully elastic relaxation. This relaxation allows the energy $e_i = p_i \cdot h_i$ of a task to be spent anywhere in the interval $[\text{est}_i, \text{lct}_i)$, regardless of the task's height or its non-preemption. For instance, on a resource of capacity 2, a task with $\text{est}_i = 0$, $\text{lct}_i = 4$, $p_i = 4$, and $h_i = 1$ could execute using 2 units of the resource at time 0, one unit at time 2 and one unit at time 3 for a total of $p_i \cdot h_i = 2 \cdot 2 = 4$ units of energy.

One of the strongest rules using the fully elastic relaxation is the energetic reasoning [2,16]. This rule, as the name suggests, is based on the notion of minimum energy in an interval $[l, u)$. The *left shift* of a task i in the interval, noted $\texttt{LS}(i, l, u) = h_i \cdot \max(0, \min(u, \text{ect}_i) - \max(l, \text{est}_i))$, is the amount of energy the task consumes in the interval when it is scheduled at its earliest. Conversely, the *right shift* of task i in interval $[l, u)$, noted $\texttt{RS}(i, l, u) = h_i \cdot \max(0, \min(u, \text{lct}_i) - \max(l, \text{lst}_i))$, is the amount of energy in the interval when the task is scheduled at its latest. The *minimum intersection* $\texttt{MI}(i, l, u) = \min(\texttt{LS}(i, l, u), \texttt{RS}(i, l, u))$ is the minimum between the left and right shift. It represents the minimum amount of energy that the task consumes in the interval regardless of when it starts. The sum of the minimum intersection of all tasks, noted $\texttt{MI}(\mathcal{I}, l, u) = \sum_{i \in \mathcal{I}} \texttt{MI}(i, l, u)$, is a fully elastic lower bound on the amount of energy consumed in an interval. The energetic reasoning detection rule (1) states that if there exists an interval such that the minimum intersection is greater than the energy

available on the resource, i.e. $C \cdot (u - l)$, the CUMULATIVE constraint cannot be satisfied. Baptiste et al. [2] showed that it is sufficient to consider a subset of $O(n^2)$ intervals said *of interest* to apply the rule.

$$\exists [l, u) \ | \ \texttt{MI}(\mathcal{I}, l, u) > C \cdot (u - l) \implies \texttt{fail} \tag{1}$$

Many industrial problems require that a resource has a minimum usage instead of, or in addition to, a maximum. This is the case for the shifts scheduling [8] and the nurse rostering [4] problems. Both problems consist in scheduling the shifts of employees or nurses to satisfy a demand while minimizing the cost of exceeding it.

2.1 Global Cardinality Constraint

The Global Cardinality Constraint [13,15,22] GCC($[X_1, \ldots, X_n], [v_1, \ldots, v_m], [l_1, \ldots, l_m], [u_1, \ldots, u_m]$) ensures that each value v_i is assigned to at least l_i and at most u_i variables in X. If all tasks have a processing time $p_i = 1$ and share the same height, the GCC can be used to model the minimum usage of a resource. The variable X_i represents the starting time of the task i. The values are the time points. When the parameters u_j are set to infinity, the GCC forces each time point to be assigned a minimum number of times. However, there is currently no global constraint to handle the general case where tasks have distinct processing times or distinct heights.

2.2 Generalized Cumulative

Beldiceanu and Carlsson [3] introduced a generalization of the CUMULATIVE constraint, presented in (2). The GENERALIZEDCUMULATIVE constraint supports tasks with negative heights and an operator op $\in \{\leq, \geq\}$ that allows the capacity of the resource to be either a maximum that must not be exceeded or a minimum that must be reached. One can use this generalization to model a problem where tasks must meet a demand vector $d = [d_0, \ldots, d_{\texttt{hor}-1}]$. The demand d_t is the minimum amount of energy that must be spent by the tasks at time t. We show how this can be done in Section 3. We describe the portion of their algorithm that ensures that a minimum is reached.

$$\begin{aligned} &\textsc{GeneralizedCumulative}(S, p, h, C, \text{op}) \overset{\text{def}}{\iff} \\ &\quad \sum_t \sum_i h_i \cdot \texttt{boolToInt}(S_i \leq t < S_i + p_i) \text{ op } C \end{aligned} \tag{2}$$

To filter the GENERALIZEDCUMULATIVE constraint, Beldiceanu and Carlsson proposed a sweep algorithm that performs a Time-Tabling reasoning. The main idea behind the algorithm is to compute an upper bound of the resource usage at each time point and check whether that upper bound satisfies the minimum capacity.

Let GOOD$(t) = \{i \in \mathcal{I} \mid h_i > 0 \wedge \text{est}_i \leq t < \text{lct}_i\}$ be the set of tasks with positive height that *can* execute at time point t. These tasks increase the usage of the resource and thus can contribute to meet the minimum capacity. Let BAD$(t) = \{i \in \mathcal{I} \mid h_i < 0 \wedge \text{lst}_i \leq t < \text{ect}_i\}$ be the set of tasks with a negative height that have a compulsory part at time point t and therefore *must* execute at that time point. These tasks decrease

the usage and thus make the minimum capacity harder to reach. The checker rule for Beldiceanu and Carlsson's algorithm is presented in (3). Recall that C is the constant representing the capacity of the resource.

$$\exists t \sum_{i \in \text{GOOD}(t)} h_i + \sum_{i \in \text{BAD}(t)} h_i < C \implies \text{Failure} \tag{3}$$

The optimistic scenario occurs where, at a time point t, all $\text{GOOD}(t)$ tasks that can execute do so, and only the $\text{BAD}(t)$ tasks that *must* execute due to their compulsory parts do so. This means that if, at any time point, the sum is not enough to satisfy the capacity, the constraint cannot be satisfied. Note that this rule does not take the processing time of the tasks into account.

The filtering rule (4) filters GOOD tasks based on this idea. Consider a task j and a time point t. If the lower bound on the resource usage at time point t is insufficient to meet the minimum capacity without the contribution of task j, then task j must be executing at time t. The domain of S_j is changed for $\text{dom}(S_j) \cap [t - p_i + 1, t]$.

$$\forall t \, \forall j \in \text{GOOD}(t) \sum_{i \in \text{GOOD}(t)} h_i + \sum_{i \in \text{BAD}(t)} h_i - h_j < C \implies \text{lst}_j \leq t < \text{ect}_j \tag{4}$$

2.3 SoftCumulative

De Clerc et al. [7] and Ouellet and Quimper [20] proposed checker and filtering algorithms for the SOFTCUMULATIVE constraint, a version of the CUMULATIVE constraint where it is possible to overload the resource but at a cost. The definition of SOFTCUMULATIVE (5) generalizes the CUMULATIVE constraint by adding the *overcost* variable Z, an upper bound on the cost incurred by overloading the resource. Note that it is not an equality. Generally, the cost is either minimized or subject to another constraint.

Ouellet and Quimper [20] introduced a generic cost function, but, for the sake of simplicity, we assume the cost function is linear in this paper. That is, overloading the resource by x units of resources always costs x units.

$$\begin{aligned} &\text{SOFTCUMULATIVE}(S, p, h, C, Z) \overset{\text{def.}}{\iff} \\ &\quad Z \geq \sum_t \max(0, \sum_i h_i \cdot \texttt{boolToInt}(S_i \leq t < S_i + p_i) - C) \end{aligned} \tag{5}$$

Ouellet and Quimper adapted the energetic reasoning for the SOFTCUMULATIVE constraint. Instead of searching for one interval with an overload, their algorithm searches for a partition of the time line into contiguous intervals such that the sum of the overload in each interval is maximized. That sum is a lower bound on the overcost. The partition can be computed in $O(|T|^2)$ steps using dynamic programming, where T is the set of time points to partition. If the set T corresponds to the lower and upper bounds of the intervals of interest, there is $O(n^2)$ time points and thus, computing the partition is in $O(n^4)$. This is not reasonable for an algorithm that is called thousands of times during the search. However, Ouellet and Quimper proposed to consider a set with only $4n$ time points, which correspond to the est, ect, lst, and lct of the tasks. By doing

so, the algorithm enforces a weaker filtering, but the complexity is better. Nevertheless, they showed that, when used on the hard version of the CUMULATIVE constraint (with $Z = 0$), the algorithm using only $4n$ time points applies the Time-Tabling and the Edge-Finding rules of the CUMULATIVE constraint.

3 Min-Cumulative

We introduce the MINCUMULATIVE constraint that ensures that the sum of the heights of the tasks in execution at each time point t is greater than or equal to the demand d_t.

$$\text{MINCUMULATIVE}(\ [S_1,\ldots,S_n],[p_1,\ldots,p_n],\ [h_1,\ldots,h_n],[d_0,\ldots,d_{\mathtt{hor}-1}]) \iff \forall t \in \mathcal{T} \sum_{i\in\mathcal{I}:S_i\leq t<S_i+p_i} h_i \geq d_t \tag{6}$$

This constraint is a special case of the GENERALIZEDCUMULATIVE constraint presented by Beldiceanu and Carlsson, but a specialized version allows the design of more specialized and stronger checker and filtering algorithms. One can model the MINCUMULATIVE with GENERALIZEDCUMULATIVE by adding fixed tasks of negative heights to the problem to represent the demand. For each interval $[l, u)$ of maximal length such that $d_t = d_{t+1}\ \forall t \in \{l..u-2\}$, we created a fixed task i with $\mathrm{est}_i = l$, $\mathrm{lct}_i = u$, $p_i = u - l$, and $h_i = -d_l$. For instance, a demand of $[1, 1, 2, 2]$ in MINCUMULATIVE can be represented by two tasks with $\mathrm{est}_1 = 0$, $\mathrm{lct}_1 = \mathrm{est}_2 = 2$, $\mathrm{lct}_2 = 4$, $p_1 = p_2 = 2$, $h_1 = -1$, and $h_2 = -2$. By fixing the capacity $C = 0$ and using operator op $= \geq$, the GENERALIZEDCUMULATIVE encodes the MINCUMULATIVE.

We can generalize the constraint where processing times and heights are variables rather than constants. The algorithms we present can substitute the height and the processing time by the maximum value in the domain of these variables.

Theorem 1. *Deciding whether the* MINCUMULATIVE *constraint admits a solution is NP-Complete even when domains are intervals.*

Proof. Deciding whether MINCUMULATIVE is feasible is a special case of the strongly NP-Complete unrestricted-output 3-Partition problem (`3-Part-U0`) [12]. Recall that `3-Part-U0` is the problem of deciding whether it is possible to partition the multiset R containing $n = 3m$ integers with a total sum of $m \cdot T$ into m multisets, each of sum T. For each integer R_i, we declare a task with $\mathrm{dom}(S_i) = [1, m]$, $p_i = 1$, $h_i = R_i$. The demand is T at each of the m time points. The starting times of the tasks correspond to the index of the set in which the matching integer is partitioned. The sum of the heights of all tasks is $m \cdot T$ and there is m time points of demand T. Hence, each of the m time points must have a usage of exactly T and corresponds to one of the m multisets. □

Since deciding the feasibility of MINCUMULATIVE is NP-Complete even when domains are intervals, propagators cannot apply either bounds or domain consistency and must instead rely on relaxed rules to detect failures or filter variables.

To enforce the MINCUMULATIVE, it is sufficient to use a decomposition with summations and inequality constraints, as in (6). This method has the disadvantage of requiring a number of constraints and variables that is function of the horizon. In problems

with a large horizon, this solution may not scale. We propose solutions that scale with the number of time points and that filters more than the decomposition.

4 Underload check

We introduce the *underload check* rule that detects when MINCUMULATIVE is unfeasible. This new rule is inspired from the overload check [27] and the fully elastic relaxation. As with the overload check, the underload check is not sufficient to enforce the MINCUMULATIVE. It needs to be paired with another rule, such as the time-tabling.

The *underload detection rule* is based on the rule for the lower-bound constraint, a special case of the GCC constraint [22] where values must occur a minimum number of times within a vector of variables. The detection rule finds a non-empty subset $U \subseteq \mathcal{T}$ of time points for which the energy of the tasks that can be spent at these time points is not enough to satisfy the demand. When such a set exists, too much energy is spent in $\mathcal{T} \setminus U$ and not enough in U. We say that $\mathcal{T} \setminus U$ is *overloaded* while U is *underloaded*.

Underload detection rule: The underload detection rule fails iff there exists a set of time points U whose demand exceeds the energy of the tasks that can be executed during U:

$$\exists U \subseteq \mathcal{T} \;\; \sum_{t \in U} d_t > \sum_{i \in \mathcal{I}: [\mathrm{est}_i, \mathrm{lct}_i) \cap U \neq \emptyset} e_i \implies \text{Failure} \tag{7}$$

To apply the rule, we design a greedy algorithm (Algorithm 2) that schedules the energy of the tasks, in non-decreasing order of lct, as early as possible. The algorithm wastes the demand at a given time point only if there is no other option. We call that waste *overflow*. Once all tasks are scheduled, time points where the demand was not met are included in the set U. If there is none, U is empty. If $U \neq \emptyset$, the constraint cannot be satisfied.

The algorithm uses the time line data structure [9] which efficiently schedules tasks according to the fully elastic relaxation. We begin by presenting a slightly modified version of the time line. Since the time line was designed for the CUMULATIVE, we replace the maximum capacity of the resource by the demand. We also prevent the algorithm from allocating energy that is not used to fulfill the demand.

The time line is first initialized with a vector of *critical time points* $\mathcal{T}^C$ that contains, in increasing order, the est and lct of the tasks without duplicates. We define a vector of demand Δ of dimension $|\mathcal{T}^C| - 1$. A critical time point $\mathcal{T}^C[j]$ has an associated demand $\Delta[j] = \sum_{k=\mathcal{T}^C[j]}^{\mathcal{T}^C[j+1]-1} d_k$ equal to the sum of the demand of the time points in the semi-open interval $[\mathcal{T}^C[j], \mathcal{T}^C[j+1])$. Two vectors, M_{est} and M_{lct} map a task i to the index of its est and lct in $\mathcal{T}^C$ such that $\mathrm{est}_i = \mathcal{T}^C[M_{\mathrm{est}}[i]]$ and $\mathrm{lct}_i = \mathcal{T}^C[M_{\mathrm{lct}}[i]]$. The time line uses a disjoint set (also called union-find) data structure [11] S upon the integers $1..|\mathcal{T}^C|$ containing the indexes of the critical time points. The operation $S.\texttt{union}(i, j)$ merges the set containing i with the set containing j. The operation $S.\texttt{findGreatest(i)}$ returns the greatest element of the set containing i. It has the same complexity as the classic `find` operation and is implemented by keeping a map of the greatest element of each set and updating it when `union` is called. The time

line supports the operation `ScheduleTask`(i) (Algorithm 1), which schedules task i as early as possible. Since the data structure is based on the fully elastic relaxation, the task can be preempted and can take more than its height at any given time point. It overflows at a time point only if it has to.

`ScheduleTask`(i) is implemented differently than in [9]. The algorithm computes the energy $e_i = h_i \cdot p_i$ that needs to be scheduled (line 1). Using the disjoint set S, line 2 finds the first time point j that is greater than or equal to est_i whose the demand is not yet satisfied. The while loop schedules as much energy as possible at that time point, but no more than the demand. If the demand of the current critical time point is fulfilled ($\Delta_t = 0$), the algorithm merges, on line 3, the sets of indices containing j with the one containing $j + 1$. This allows for future calls to `S.findGreatest` to return the next time point with unfulfilled demand. The process is repeated until all the task's energy is scheduled or the demand in the interval $[\mathrm{est}_i, \mathrm{lct}_i)$ is fulfilled.

Algorithm 1: ScheduleTask(i)

1 $e \leftarrow h_i \cdot p_i$;
$j \leftarrow M_{\mathrm{est}}[i]$;
$t_1 \leftarrow -\infty$;
while $e > 0 \wedge t_1 < \mathrm{lct}_i$ **do**
2 $\quad j \leftarrow S$.`findGreatest`(j);
$\quad t_1 \leftarrow \mathcal{T}^C[j]$;
$\quad \tau \leftarrow \min(\Delta[j], e)$;
$\quad e \leftarrow e - \tau$;
$\quad \Delta[j] \leftarrow \Delta[j] - \tau$;
$\quad$ **if** $\Delta[j] = 0$ **then**
3 $\quad\quad S$.union$(j, j + 1)$;

Algorithm 2: UnderloadCheck($\mathcal{I}, d$)

InitializeTimeline($\mathcal{I}, d$);
for $i \in \mathcal{I}$ *sorted by* *non-decreasing* lct **do**
$\quad$ ScheduleTask(i);
1 **return** S.`findGreatest(1)` $= |\mathcal{T}^C|$;

We present the UnderloadCheck (Algorithm 2). Once the time line is initialized, the algorithm schedules each task in order of non-decreasing lct. The algorithm greedily spends the task's energy $e = p_i \cdot h_i$ as early as possible, but no more than the demand at each time point. If the loop stops because of the condition $t_1 < \mathrm{lct}$, the unscheduled energy of the task cannot be used to fill the demand, because it would be scheduled after the latest completion time of the task. Thus, the energy is lost and we say that the task *overflowed*. Once all tasks are processed, the algorithm checks, on line 1, if all the sets have been merged together. If so, the demand of all time points is met and the check passes. Otherwise, the demand of at least one time point is not met and the check fails.

Example 1. Recall a task is defined by $\langle i, \mathrm{est}_i, \mathrm{lct}_i, p_i, h_i \rangle$. Given the three tasks $\langle 1, 2, 4, 2, 1 \rangle$, $\langle 2, 2, 5, 2, 1 \rangle$, $\langle 3, 0, 6, 2, 1 \rangle$ and a demand of 1 for each time point in $[0, 6)$, the `UnderloadCheck` computes the vector $\mathcal{T}^C = [0, 2, 4, 5, 6]$ of critical time points. In the disjoint sets data structure S, all these time points are in separate sets: $\{0\}, \{2\}, \{4\}, \{5\}, \{6\}$ (in this example, we use the time point values instead of indexes for the sake of clarity). The demand vector is initialized to $\Delta = [2, 2, 1, 1]$.

The algorithm begins by scheduling task 1, which has the smallest lct, in interval $[2, 4)$ (see Figure 1). Since all the demand for critical time point 2 is met, the disjoint set $\{2\}$ is merged with disjoint set $\{4\}$. The `UnderloadCheck` then processes task 2,

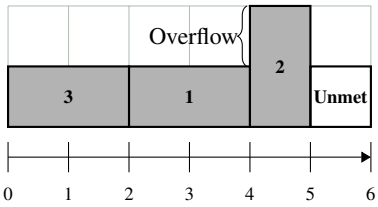

Fig. 1: Visual representation of the time line after the execution of the `UnderloadCheck`. One unit of demand has not been met in $[5, 6)$.

which has the second smallest lct. The algorithm finds the greatest time point greater than or equal to $\mathrm{est}_2 = 2$, which is 4. Hence, it schedules one unit at time point 4 and merges the set $\{2, 4\}$ with $\{5\}$. However, since $\mathrm{lct}_2 = 5$, the remaining unit of energy of task 2 is lost, causing an overload. The `UnderloadCheck` then processes the last task, 3. As the greatest time point greater than or equal to $\mathrm{est}_3 = 0$ is 0, the algorithm schedules 2 units of energy of task 3 in the interval $[0, 2)$, before merging the set $\{0\}$ with $\{2, 4, 5\}$. Once all tasks have been scheduled, we have $S = \{0, 2, 4, 5\}, \{6\}$. There is more than one disjoint set in S, which means that the demand has not been completely met, leading to a failure.

Theorem 2. *`UnderloadCheck` returns* true *if and only if the underload detection rule* (7) *does not fail.*

Proof. ($\Longrightarrow$) When the algorithm decrements $\Delta[j]$, it schedules the task i to spend its energy within $[\mathcal{T}^C[j], \mathcal{T}^C[j+1]) \subseteq [\mathrm{est}_i, \mathrm{lct}_i)$. Once all tasks are scheduled, if $\Delta = \vec{0}$ (or equivalently $S.\texttt{findGreatest(1)} = |\mathcal{T}^C|$), all the demand is satisfied and the elastic schedule built by the algorithm disproves the existence of a non-empty set U that satisfies the underload check detection rule.

($\Longleftarrow$) If the algorithm returns *false*, we show that there exists a set U that satisfies the condition in (7). We associate a time interval τ_i to every task i. If task i does not overflow, we set $\tau_i = \emptyset$. If task i overflows, the critical index $M_{\mathrm{lct}}[i]$ belongs to a set A in S forming an interval $\tau_i = [\mathcal{T}^C[\min(A)], \mathcal{T}^C[\max(A)])$. The demand is completely fulfilled in the time interval τ_i since $\Delta[j] = 0$ for all critical points $\mathcal{T}^C[j] \in \tau_i$. Whether task i overflows or not, we have $\sum_{j:[\mathrm{est}_j, \mathrm{lct}_j) \subseteq \tau_i} e_j \geq \sum_{t \in \tau_i} d_t$.

Let $U = [1, m] \setminus \bigcup_{i \in \mathcal{I}} \tau_i$. Consider a task j such that $[\mathrm{est}_j, \mathrm{lct}_j) \cap U \neq \emptyset$. Its energy was fully spent inside U. Indeed, all the non-empty intervals τ_i such that $\mathrm{lct}_i < \mathrm{lct}_j$ were fulfilled before j was processed. Its energy was therefore spent before or after τ_i. For non-empty intervals τ_i such that $\mathrm{lct}_i \geq \mathrm{lct}_j$, we necessarily have $\mathrm{est}_j < \min(\tau_i)$ and since the task is scheduled at its earliest, it was first scheduled outside of τ_i. Moreover, it could not be scheduled in τ_i as the time point $M_{\mathrm{est}}[i]$ would have been merged to the set A that defined τ_i, which contradicts $\mathrm{est}_j < \min(\tau_i)$. Therefore, none of the energy of j was scheduled within τ_i. Since all the energy of the tasks j such that $[\mathrm{est}_j, \mathrm{lct}_j) \cap U \neq \emptyset$ was scheduled in U and that this energy does not fulfill the demand, the condition in (7) holds. □

Theorem 3. *`UnderloadCheck` has a complexity of $\Theta(n)$ provided that the tasks are already sorted by their latest completion times.*

Proof. Fahimi et al. [9] showed that `ScheduleTask` executes in $O(1)$ amortized time when $|\mathcal{T}^C| \in O(n)$. As our modifications only make it run faster since the energy of a task is not always fully scheduled, the complexity remains unchanged. `ScheduleTask` is called n times hence $\Theta(n)$. □

5 Underload Filtering

We can derive a filtering rule from the `UnderloadCheck` by fixing the starting time of a task, as shown in (8). If the `UnderloadCheck` fails when task i starts at time t, then task i obviously cannot start at time t and we can remove t from $\mathrm{dom}(S_i)$.

$$\neg\texttt{UnderloadCheck}(\mathcal{I} \setminus \{i\} \cup \{\langle t, t+p_i, p_i, h_i\rangle\}) \implies S_i \neq t \tag{8}$$

By studying the set of time points U that makes the underload check fail, one can derive a filtering rule that prunes more than a single value from the domain of S_i.

$$\begin{aligned} &\exists U \subseteq \mathcal{T} : D > E \\ &\qquad\qquad \implies \mathrm{ect}_i \geq \min\{t \in U \mid t \geq \mathrm{ect}_i\} + \left\lceil \frac{D-E}{h_i} \right\rceil \\ &\textbf{where } D = \sum_{t \in U} d_t \\ &\qquad\quad E = \sum_{j \in \mathcal{I}\setminus\{i\} : [\mathrm{est}_j, \mathrm{lct}_j) \cap U \neq \emptyset} e_j \end{aligned} \tag{9}$$

To filter a given task i, the rule (9) uses a set of time points U for which the sum of the demand $D = \sum_{t \in U} d_t$ exceeds the energy $E = \sum_{j \in \mathcal{I}\setminus\{i\} : [\mathrm{est}_j, \mathrm{lct}_j) \cap U \neq \emptyset} e_j$ of the tasks that can spend energy in U (except task i). If the energy of the tasks without i is insufficient to cover the demand for U by $D - E$ units, then task i must spend $D - E$ units of energy in U. Thus, we can filter the earliest completion time such that i ends at its earliest in U by the missing units.

5.1 Naive algorithm

Algorithm 3 naively applies rule (8). For each task i, it fixes the starting time to est_i, then it executes the `UnderloadCheck`. If the check fails, it increases the starting time by one and executes the `UnderloadCheck` again. It repeats until it finds a starting time t for which the check passes or there is a starting time that exceeds the latest starting time of the task. In the first case, it filters the earliest starting time of task i to t. In the second case, it returns a failure. Filtering the latest completion time is symmetric.

This algorithm can fail even if the `UnderloadCheck` passes since the filtering algorithm, contrary to the `UnderloadCheck`, fixes the current task, preventing it from having elastic energy. Hence, the check performed by the filtering algorithm is stronger.

Algorithm 3: NaiveFiltering($\mathcal{I}$, D)

```
for i ∈ I do
    t ← lct_i;
    lct_i ← ect_i //Temporarily fix i to its earliest;
    while est_i ≤ lst_i ∧
     ¬UnderloadCheck(I) do
        est_i ← est_i +1;
        lct_i ← lct_i +1;
    if est_i ≤ lst_i then
        lct_i ← t //Restore i to its original lct_i;
    else
        return false // Failure;
return true // Consistent;
```

We call this algorithm naive because, although it is simple, it requires for each task, in the worst case, $\mathrm{lst}_i - \mathrm{est}_i$ calls to the `UnderloadCheck`, leading to a worst-case time complexity of $O(|\mathcal{T}| \cdot n^2)$, which depends on the number of time points. However, this does not make the algorithm irrelevant in practice. The best-case complexity is $\Theta(n^2)$ and it happens when there is no filtering done. Furthermore, for a given task i, the algorithm executes the `UnderloadCheck` $1 + k$ times, where k is the number of values removed from the domain of S. Hence, the complexity of the algorithm increases only if it filters, which is generally not frequent though essential.

5.2 Overflow algorithm

We improve upon the naive algorithm by filtering the earliest starting time of a task i by more than one unit at a time, allowing the algorithm to directly apply rule (9). Let $\delta = \sum_{j \in \mathcal{T}^C | \mathcal{T}^C_j \geq est_i} \Delta[\mathcal{T}^C_j]$ be the amount of demand not met at or after est_i after a call to `UnderloadCheck`. Let j be the task with the smallest est_j that overflowed such that $\mathrm{est}_j \geq \mathrm{est}_i$. If such a task exists, let $\tau = \mathrm{est}_j$ otherwise let $\tau = \mathrm{est}_i$. The starting time of i can be filtered such that $\mathrm{est}_i \geq \tau + \left\lceil \frac{\delta}{h_i} \right\rceil$.

We now explain the intuition behind that filtering. In the first case, $\tau = \mathrm{est}_i$. We know that there are δ units to fill after est_i. Moving the task by one unit can *free* at most h_i units of energy (it can come from the overflow of either i or another task that will its place). We need to move i by $\left\lceil \frac{\delta}{h_i} \right\rceil$ to gain enough energy to satisfy the demand.

The second case occurs when $\tau > \mathrm{est}_i$. In that case, we move i after the est_j of the task that overflowed, by an amount corresponding to the missing energy δ. This allows j to take the place of i. The energy of i can then be used to fulfill the missing demand.

Example 2. Given two tasks $\langle 1, 0, 6, 3, 1 \rangle$ and $\langle 2, 0, 4, 3, 1 \rangle$ and a demand of 1 for each time point in $[0, 6)$, the overflow algorithm begins by filtering task 1. It temporarily fixes task 1 such that $\mathrm{lct}_1 = 3$ before running the `UnderloadCheck`. Since the temporary

lct_1 is smaller than $\mathrm{lct}_2 = 4$, the `UnderloadCheck` starts by scheduling task 1 in $[0, 3)$ (see Figure 2). Then, it schedules task 2. There are 3 units of energy to schedule and the task can only be scheduled in $[0, 4)$. The algorithm schedules one unit at time 3 and the two remaining units are wasted. All tasks are scheduled, but two units of demand are not met ($\delta = 2$), as shown on Figure 2. Since the `UnderloadCheck` fails, the overflow algorithm filters task 1. As task 2 overflowed, we have $\tau = \mathrm{est}_2 = 0$. The algorithm filters est_1 to $\tau + \left\lceil \frac{\delta}{h_1} \right\rceil = 0 + \frac{2}{1} = 2$. Hence, the overflow algorithm increased the est_1 by two units with a single call to `UnderloadCheck`.

The algorithm iteratively applies this rule until the `UnderloadCheck` passes (or a failure is detected). Its worst-case complexity is as the naive algorithm's but it increments the est_i by more than one unit, reducing the number of calls to the `UnderloadCheck`.

6 Model with SoftCumulative

We show how to encode the MINCUMULATIVE using the SOFTCUMULATIVE. By doing so, we can use the strong energetic reasoning rules from the algorithms introduced by Ouellet and Quimper [20] for the SOFTCUMULATIVE.

Let $\mathcal{I}'$ be a set of fixed tasks that cover the horizon. For each interval $[l, u)$ of maximal length such that $d_j = d_{j+1}\ \forall j \in \{l..u-2\}$, we create a task $i \in \mathcal{I}'$ with $\mathrm{est}_i = l$, $\mathrm{lct}_i = u$, $p_i = u - l$, and $h_i = \max_t(d_t) - d_l$. We have this equivalence.

$$\begin{aligned} &\textsc{MinCumulative}([S_i \mid i \in \mathcal{I}], [p_i \mid i \in \mathcal{I}], [h_i \mid i \in \mathcal{I}], d) \iff \\ &\textsc{SoftCumulative}([S_i \mid i \in \mathcal{I} \cup \mathcal{I}'], [p_i \mid i \in \mathcal{I} \cup \mathcal{I}'], [h_i \mid i \in \mathcal{I} \cup \mathcal{I}'], \\ &\qquad \max_t(d_t), \sum_i e_i - \sum_t d_t) \end{aligned} \tag{10}$$

Lemma 1. *The* MINCUMULATIVE *can be encoded as a* SOFTCUMULATIVE *as in* (10).

Proof. The SOFTCUMULATIVE is associated to a resource of capacity $\max_t(d_t)$ over a horizon `hor`. The constraint allows an overflow of at most $\sum_{i \in \mathcal{I}} e_i - \sum_t d_t$ units of

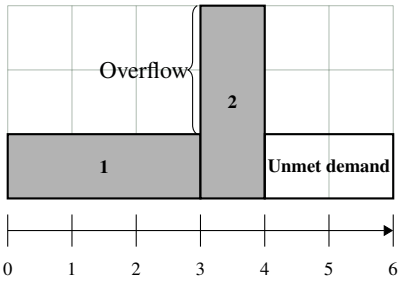

Fig. 2: Visual representation of the time line after the first call to the `UnderloadCheck` by the overflow algorithm, when filtering task 1. Task 1 is scheduled in $[0, 3)$. Task 2 has one unit scheduled in $[3, 4)$ and two units overflowed. Two units of demand are not met in $[4, 6)$.

energy. So at most $\max_t(d_t) \cdot \texttt{hor} + \sum_{i \in \mathcal{I}} e_i - \sum_t d_t$ units of energy can be scheduled. The energy of the tasks matches this bound meaning that the resource is fully used.

$$\sum_{i \in \mathcal{I}} e_i + \sum_{i \in \mathcal{I}'} e_i = \sum_{i \in \mathcal{I}} e_i + \sum_l (\max_t(d_t) - d_l) = \sum_{i \in \mathcal{I}} e_i + \texttt{hor} \cdot \max_t(d_t) - \sum_l d_l$$

Since the tasks in $\mathcal{I}'$ consume $\max_t(d_t) - d_l$ units of energy at time l, the tasks in $\mathcal{I}$ consume the remaining d_l (and can overflow) as required by the MINCUMULATIVE. The converse holds using the same argument. □

7 Comparing the rules

We compare the check rules of the Time-Tabling (TT), the Underload Check (UC), and the Energetic Reasoning (ER). The Underload Check rule, as mentioned earlier, is insufficient to enforce the MINCUMULATIVE constraint, as illustrated in Example 3.

Example 3. Consider an instance with two fixed tasks such that $\text{est}_1 = 0$, $\text{est}_2 = 1$, $\text{lct}_1 = 2$, $\text{lct}_2 = 3$, $p_{1,2} = 2$, $h_{1,2} = 1$ and a demand $d = [1, 1, 2]$. The Underload Check rule finds no underloaded set since it schedules the first unit of energy of task 1 at time point 1, the second unit at time point 2 and the two units of task 2 at time point 3. Even when the tasks are fixed, the Underload Check relaxes the heights of the tasks. On the other hand, the Time-Tabling finds that there are not enough tasks at time point 3 to satisfy the demand of 2. Similarly, the energetic reasoning finds that the minimum intersection in the interval $[1, 3)$ is 3. Since the demand in the interval is 2, there is one unit of overcost. The sum of the energy of the tasks is 4 and the sum of the demand is 4 so the SOFTCUMULATIVE allows no overcost, hence the detection of the failure.

While the weakness of the Underload Check is to relax the height of the tasks, the weakness of the Time-Tabling is that it does not take the processing time of the tasks into account, as demonstrated by Example 4

Example 4. Consider a task ($\text{est}_1 = 0$, $\text{lct}_1 = 5$, $p_1 = 1$, $h_1 = 1$) and a demand $d = [1, 1, 1, 1]$. The Time-Tabling notices that task 1 can cover the demand of each time point individually and passes. The Underload Check and the Energetic Reasoning quickly find that there is not enough energy in task 1 to satisfy the overall demand.

Examples 3 and 4 show that the Underload Check and the Time-Tabling are incomparable when comes the time to detect infeasibility, as it is the case for the CUMULATIVE. Lemma 2 shows that the energetic reasoning is stronger than the Time-Tabling.

Lemma 2. *If the Time-Tabling checker fails, then the* SOFTCUMULATIVE *model using the energetic reasoning checker also fails.*

Proof. Suppose the Time-Tabling fails because the total height of the tasks at time t is less than d_t. Consider the partition $\mathcal{T} = [0, t) \cup [t, t + 1) \cup [t + 1, \texttt{hor})$. From Lemma 1, the resource is fully used. If it is underused in interval $[t, t+1)$, it is overused in $[0, t) \cup [t+1, \texttt{hor})$. The energetic reasoning will therefore detect an overflow greater than $\sum_i e_i - \sum_t d_t$ in the intervals $[0, t) \cup [t + 1, \texttt{hor})$ and thus, detect failure. □

Lemma 3. *If the Underload Check fails, then the* SOFTCUMULATIVE *model using the energetic reasoning checker also fails.*

Proof. Suppose that the Underload Check returns a failure when checking the set U. The same reasoning as with Lemma 2 applies; the intervals in $\mathcal{T} \setminus U$ have an overcost greater than $\sum_i e_i - \sum_t d_t$. Thus the SOFTCUMULATIVE fails. □

The SOFTCUMULATIVE with its energetic checker is stronger than the Underload Check and Time-Tabling. In fact, it is strictly stronger, as shown in Example 5.

Example 5. Consider a demand $d = [1, 3, 1]$ and two tasks: $\text{est}_1 = \text{est}_2 = 0$, $\text{lct}_1 = 2$, $\text{lct}_2 = 3$, $p_1 = 2$, $p_2 = 1$, $h_1 = 2$, and $h_2 = 1$. The Time-Tabling passes since, when considered individually, there can be three units of height at time points 0 and 1 and one unit at time point 2. The Underload Check passes since the four units of energy of task 1 cover the demand of the first two time points and the single unit of task 2 covers the demand of the last time point. However, the SOFTCUMULATIVE model fails. Indeed, we have $MI(\mathcal{I}, 0, 1) = 2$, but the demand in $[0, 1)$ is only 1. Hence, we have an overcost of 1 in that interval, but no overcost is allowed since $\sum_{i\in\mathcal{I}} e_i - \sum_t d_t = (4+1) - 5 = 0$. Hence the SOFTCUMULATIVE detects the failure.

8 Experiments

We tested our algorithm on two benchmarks. The first contains randomly generated instances of a simple problem with one MINCUMULATIVE constraint. The tasks have a variable height in $\{0, 1\}$ indicating whether a task is activated or not. The goal is to satisfy the demand while minimizing the number of activated tasks. The instances are available on Github[1] and upon request to the second author. The Work Shift Scheduling Benchmark is an industrial benchmark introduced by [6]. The problem consists in scheduling the work shifts of employees while satisfying the demand. The benchmark has instances with up to 10 activities, but we only use the instances with one activity. Employee $e \in E$ can work between 6 to 8 hours. An employee e starts at time $S_{e,1}$, takes a 15-minute break, resumes at S_{e_2}, takes a 1-hour lunch, resumes at $S_{e,3}$, takes a 15-minute break, and resumes at $S_{e,4}$. The 4 work periods are at least one hour. The demand can vary at each 15-minute time step. We minimize the number of employees. The model for the decomposition of our constraint in smaller binary constraints is given in (11)-(17). We replace (15) by the MINCUMULATIVE constraint for the other configurations. We break symmetries with (16) and (17). The upper limit on the number of employees is set to $|E| = 10$. This gives two unsatisfiable instances, which allows us to observe the behaviour of the algorithms on both satisfiable and unsatisfiable instances. The search heuristic branches on the minimum value of H_e, then P_e, and finally $S_{e,p}$.

$$\text{minimize } \textstyle\sum_{e\in E} H_e \text{ subject to} \tag{11}$$

$$S_{e,p} + P_{e,p} + \delta = S_{e,p+1} \qquad \forall e \in E, (p, \delta) \in \{(1, 1), (2, 4), (3, 1)\} \tag{12}$$

$$S_{e,4} + P_{e,4} + 4 \leq |T| \qquad \forall e \in E \tag{13}$$

[1] https://github.com/yanickouellet/min-cumulative-paper-public

$$24 \leq \sum_{p \in \{1..4\}} P_{e,p} \leq 32 \qquad \forall e \in E \quad (14)$$

$$\sum_{e \in E, p \in \{1..4\}} H_e \cdot (S_{e,p} \leq t < S_{e,p} + P_{e,p}) \geq d_t \qquad \forall t \in T \quad (15)$$

$$H_e \leq H_{e-1} \wedge S_{e-1,2} \leq S_{e,2} \qquad \forall e \in 1..|E| - 1 \quad (16)$$

$$H_e = 0 \implies S_{e,p} = s \qquad \forall e \in E, (p,s) \in \{(1,0),(2,5),(3,13),(4,18)\} \quad (17)$$

We implemented our algorithms in Java using Choco solver version 4.10.6 [21]. We ran the experiments on an Intel Xeon Silver 4110 (2.10Ghz). All models were implemented in MiniZinc. Experiments for the random benchmark were done with a timeout of 20 minutes. Since there are fewer instances, experiments for the Work Shift Scheduling Benchmark were done with a timeout of 1 hour.

We report the time taken to solve instances of the Work Shift Scheduling Benchmark in Table 1. We experimented two versions of the problem, one with variable processing time and one with processing times fixed to 6 periods. For solved instances, the difference between the algorithms were similar, but we report only the latter since few instances were solved to optimality in the former. We tested the following configurations: the Decomposition (D), Time-Tabling filtering (TT), Time-Tabling with Underload Check (TT + UC), Time-Tabling with Underload Filtering (TT + UF), Time-Tabling with Energetic Reasoning Check (TT + EC) and Energetic Filtering (EF). We did not test the Underload algorithms alone since they are not sufficient to enforce the MINCUMULATIVE. Note that instances 7 and 10 are unsatisfiable instances.

The decomposition (D) is the configuration with the slowest solving times and it can solve to optimality only half of the instances within one hour. Time-Tabling alone is faster, but worse than when it is combined with Underload Check or Underload Filtering. The combination of Time-Tabling and Underload Check is clearly the best configuration. It solves all instances to optimality and is faster on every instance than all other configurations. The combination of Time-Tabling and Underload Filtering is the second-best configuration. However, the added cost of filtering does not seem to be worth the reduction in the search space it provides. Both Energetic Reasoning configurations are slower on this benchmark. They reduce the search space more than the Underload Check configurations, but their high complexity is not worth it.

We report the results of the random benchmark in Table 2. The first number in the name of each instance indicates the number of tasks. We report only instances for which at least one configuration found the optimality within 20 minutes. We can see that the Decomposition and Time-Tabling struggle on this benchmark, having difficulties solving instances of more than 20 tasks. On smaller instances, Time-Tabling combined with Underload Check performs better than other configurations, while, on larger instances, Time-Tabling combined with Energetic Check is better. The latter is the only configuration able to solve all instances. We conclude that, as the problem becomes harder, the combination of Time-Tabling and Energetic Checker becomes more interesting, even with a slower complexity. However, as for the Work Shift Scheduling Benchmark, the added cost of the Energetic Filtering is not worth the increased filtering.

9 Conclusion

The MINCUMULATIVE enforces tasks to cover a minimum demand. It is NP-Complete to test for feasibility. We proposed a checker algorithm, the `UnderloadCheck`, and two filtering algorithms based on the checker: the Naive filtering algorithm, and the Overflow filtering algorithm. MINCUMULATIVE can be encoded with a SOFTCUMULATIVE. We compared the strength of the different checking rules. The combination of Time-Tabling and Underload Check performs best in practice.

Instance	**D**		**TT**		**TT + UC**		**TT + UF**		**TT + EC**		**EF**	
	time (s)	bt	time (s)	bt	time (s)	bt	time (s)	bt	time (s)	bt	time (s)	bt
1	3.7	5.5	0.8	6.3	0.7	3.3	0.8	1.6	1.6	3.3	3.3	2.3
2	729.8	139.9	45.4	1,528.2	4.4	128.9	15.6	86.7	39.2	93.3	131.7	54.6
3	-	-	3,106.7	103,275.4	560.7	21,395.7	2,798.7	16.3	-	-	-	-
4	-	-	2,503.2	114,540.2	242.3	9,592.2	1,035.7	6,564.2	1,637.4	3,668.9	-	-
5	542.3	1,083.1	32.0	1,154.5	6.7	221.0	23.7	145.1	54.6	146.8	188.3	77.2
6	-	-	945.2	41,763.3	144.9	5,635.7	671.6	4,067.2	2,731,462.0	1.1	-	-
7	-	-	-	-	488.3	20,286.2	2,140.0	14,867.0	1,255.1	3,928.8	-	
8	87.0	146.0	6.2	156.3	1.7	30.4	4.0	18.4	9.7	27.1	29.5	15.2
9	1.4	550.0	0.2	0.7	0.3	0.3	0.4	0.2	0.5	0.3	0.6	0.2
10	-	-	-	-	28.8	1,113.0	114.5	649.9	44.7	144.4	181.2	57.5

Table 1: Time (s) and thousands of backtracks (bt) to optimally solve the work shift scheduling problem. A dash (-) means that the optimal was not proved within 1 hour. Instances 7 and 10 are unsatisfiable.

Instance	**D**		**TT**		**TT + UC**		**TT + UF**		**TT + EC**		**EF**	
	time (s)	bt	time (s)	bt	time (s)	bt	time (s)	bt	time (s)	bt	time (s)	bt
20_1	87.4	3,776.6	125.5	12,900.0	0.3	12.6	0.9	12.6	1.3	8.7	3.9	8.4
20_2	0.5	0.9	0.4	7.9	0.1	2.4	0.4	2.4	0.2	1.3	1.3	1.2
20_3	0.3	0.3	0.2	1.1	0.1	1.0	0.3	1.0	0.2	0.8	0.9	0.8
20_4	31.8	1,463.6	248.0	20,035.1	0.2	6.2	0.5	6.3	0.4	3.7	2.3	2.8
20_5	512.9	2,1434.8	1,205.6	138,626.8	0.2	2.2	0.4	2.2	0.3	2.0	455.0	1,080.3
20_6	0.0	0.1	0.2	7.1	0.4	8.5	1.1	8.5	0.7	7.4	3.6	7.3
20_7	0.2	0.5	0.9	34.8	0.6	26.6	2.1	26.2	0.9	9.0	3.9	7.7
20_8	0.1	0.2	0.2	2.0	0.2	2.1	0.4	2.1	0.3	1.8	1.0	1.8
20_9	0.1	1.0	0.5	25.1	0.5	30.2	0.8	8.3	1.3	6.7	3.6	7.0
20_10	4.0	170.3	7.1	302.1	0.3	1.5	0.4	1.5	0.4	1.1	1.3	1.1
30_1	-	-	-	-	3.6	309.4	29.0	285.1	8.4	5.4	119.3	52.7
30_2	-	-	-	-	2.1	210.5	16.3	204.1	9.1	68.6	110.8	53.4
30_3	15.8	523.1	-	-	-	-	-	-	3.5	25.6	44.1	19.7
30_4	-	-	-	-	0.4	7.8	2.2	7.7	0.9	4.4	6.8	4.2
30_5	-	-	-	-	0.8	27.7	4.2	27.7	0.5	2.3	4.3	2.3
30_6	-	-	-	-	1,390.1	142,143.5	-	-	213.8	2,340.3	43.7	23.6
30_7	-	-	-	-	20.4	1,393.0	178.4	1,371.7	23.7	186.0	263.6	181.9
30_8	-	-	-	-	2.1	116.4	16.1	115.9	13.3	97.1	178.4	94.1
30_9	-	-	-	-	2.5	151.9	18.3	151.5	17.3	147.2	231.3	143.2
30_10	2.2	59.0	406.0	17,834.6	5.9	336.9	34.1	336.7	7.6	42.5	67.5	39.4
40_1	-	-	-	-	-	-	-	-	668.9	3,028.2	-	-
40_2	-	-	-	-	724.1	35,924.8	-	-	36.5	106.5	435.6	77.3
40_3	-	-	-	-	1,973.2	191,817.2	-	-	1,423.0	6,603.5	-	-
40_4	-	-	-	-	-	-	-	-	4.1	11.6	59.0	10.1
40_5	-	-	-	-	1,312.8	127,012.2	-	-	1,112.2	4,452.5	-	-
40_6	-	-	-	-	-	-	-	-	9.7	33.6	108.3	21.3
40_7	-	-	-	-	-	-	-	-	2,573.3	8,849.2	-	-
40_8	-	-	-	-	-	-	-	-	2,899.0	12,032.3	-	-

Table 2: Time (s) and thousands of backtracks (bt) to optimally solve random instances. (-) means the optimal solution was not proved within 20 minutes. The first number in the name of the instance is the number of tasks.

References

1. Aggoun, A., Beldiceanu, N.: Extending chip in order to solve complex scheduling and placement problems. Mathematical and computer modelling **17**(7), 57–73 (1993)
2. Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-Based Scheduling. Kluwer Academic Publishers (2001)
3. Beldiceanu, N., Carlsson, M.: A new multi-resource cumulatives constraint with negative heights. In: International Conference on Principles and Practice of Constraint Programming. pp. 63–79. Springer (2002)
4. Burke, E.K., De Causmaecker, P., Berghe, G.V., Van Landeghem, H.: The state of the art of nurse rostering. Journal of scheduling **7**(6), 441–499 (2004)
5. Carlier, J., Sahli, A., Jouglet, A., Pinson, E.: A faster checker of the energetic reasoning for the cumulative scheduling problem. International Journal of Production Research pp. 1–16 (2021)
6. Côté, M.C., Gendron, B., Quimper, C.G., Rousseau, L.M.: Formal languages for integer programming modeling of shift scheduling problems. Constraints **16**(1), 54–76 (2011)
7. De Clercq, A., Petit, T., Beldiceanu, N., Jussien, N.: A soft constraint for cumulative problems with over-loads of resource. In: Doctoral Programme of the 16th International Conference on Principles and Practice of Constraint Programming (CP'10). pp. 49–54 (2010)
8. Ernst, A.T., Jiang, H., Krishnamoorthy, M., Owens, B., Sier, D.: An annotated bibliography of personnel scheduling and rostering. Annals of Operations Research **127**(1-4), 21–144 (2004)
9. Fahimi, H., Ouellet, Y., Quimper, C.G.: Linear-time filtering algorithms for the disjunctive constraint and a quadratic filtering algorithm for the cumulative not-first not-last. Constraints **23**(3), 272–293 (2018)
10. Feydy, T., Stuckey, P.J.: Lazy clause generation reengineered. In: International Conference on Principles and Practice of Constraint Programming. pp. 352–366. Springer (2009)
11. Gabow, H.N., Tarjan, R.E.: A linear-time algorithm for a special case of disjoint set union. Journal of computer and system sciences **30**(2), 209–221 (1985)
12. Garey, M.R., Johnson, D.S.: Computers and intractability, vol. 174. freeman San Francisco (1979)
13. Jean-Charles, R.E.: Generalized arc consistency for global cardinality constraint. American Association for Artificial Intelligence (AAAI'96) pp. 209–215 (1996)
14. Kameugne, R., Fotso, L.P., Scott, J., Ngo-Kateu, Y.: A quadratic edge-finding filtering algorithm for cumulative resource constraints. Constraints **19**(3), 243–269 (2014)
15. Katriel, I., Thiel, S.: Complete bound consistency for the global cardinality constraint. Constraints **10**(3), 191–217 (2005)
16. Lopez, P., Esquirol, P.: Consistency enforcing in scheduling: A general formulation based on energetic reasoning. In: 5th International Workshop on Project Management and Scheduling (PMS'96) (1996)
17. Mercier, L., Van Hentenryck, P.: Edge finding for cumulative scheduling. INFORMS Journal on Computing **20**(1), 143–153 (2008)
18. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation via lazy clause generation. Constraints **14**(3), 357–391 (2009)
19. Ouellet, Y., Quimper, C.G.: A $o(n \log^2 n)$ checker and $o(n^2 \log n)$ filtering algorithm for the energetic reasoning. In: International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research. pp. 477–494. Springer (2018)
20. Ouellet, Y., Quimper, C.G.: The softcumulative constraint with quadratic penalty. In: to appear in AAAI Conference on Artifical Intelligence proceeding (2022)

21. Prud'homme, C., Fages, J.G., Lorca, X.: Choco solver documentation. TASC, INRIA Rennes, LINA CNRS UMR **6241** (2016)
22. Quimper, C.G., Golynski, A., López-Ortiz, A., Van Beek, P.: An efficient bounds consistency algorithm for the global cardinality constraint. Constraints **10**(2), 115–135 (2005)
23. Schutt, A., Feydy, T., Stuckey, P.J.: Explaining time-table-edge-finding propagation for the cumulative resource constraint. In: International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems. pp. 234–250. Springer (2013)
24. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Explaining the cumulative propagator. Constraints **16**(3), 250–282 (2011)
25. Tesch, A.: A nearly exact propagation algorithm for energetic reasoning in $o(n^2 \log n)$. In: International Conference on Principles and Practice of Constraint Programming. pp. 493–519. Springer (2016)
26. Vilím, P.: Timetable edge finding filtering algorithm for discrete cumulative resources. In: International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems. pp. 230–245. Springer (2011)
27. Wolf, A., Schrader, G.: $O(n \log n)$ overload checking for the cumulative constraint and its application. In: INAP. vol. 4369, pp. 88–101. Springer (2005)