The SoftCumulative Constraint with Quadratic Penalty

Yanick Ouellet, Claude-Guy Quimper

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

Abstract

The CUMULATIVE constraint greatly contributes to the success of constraint programming at solving scheduling problems. SOFTCUMULATIVE, a version of the CUMULATIVE constraint where overloading the resource incurs a penalty is, however, less studied. We introduce a checker and a filtering algorithm for SOFTCUMULATIVE, which are inspired by the energetic reasoning rule for the CUMULATIVE. Both algorithms can be used with a classic linear penalty function, but also with a quadratic penalty function, where the penalty of overloading the resource increases quadratically with the amount of the overload. We show that these algorithms are more general than existing algorithms and outperform a decomposition of SOFTCUMULATIVE in practice.

1 Introduction

Scheduling problems where tasks share a finite amount of resources are everywhere in the industry. For instance, a university might want to schedule courses in classrooms or a factory might need to plan its production to minimize the peak in the power usage of its machines. The constraint programming community invested significant efforts in developing the CUMULATIVE constraint to help solve problems where the capacity of a resource can never be overloaded. However, a less studied but as important problem is to allow the resource to be overloaded in exchange of a penalty. For instance, a pharmaceutical company could ask its workers to work overtime to ensure that enough doses of a COVID vaccine are produced before a deadline.

We present a checker and a filtering algorithm for the SOFTCUMULATIVE constraint, a generalization of the well known CUMULATIVE that allows resource overloads. Our algorithms are based on the energetic reasoning rule used by CUMULATIVE. The algorithms work for both linear and quadratic penalty functions. To the best of our knowledge, a quadratic penalty function cannot currently be modelled by using an existing global constraint.

Section 2 presents a background from the literature. We introduce our version of SOFTCUMULATIVE (Section 3), present our checker (Section 4) and filtering (Section 5) algorithms. We explain how to use our algorithms with lazy clause generation solvers in Section 6. Section 7 shows how relevant our algorithms are in practice before concluding.

2 Background

Scheduling

Consider a set $\mathcal{I}$ of n tasks. Each task $i \in \mathcal{I}$ has to be executed between its *earliest starting time* est_i and its *latest completion time* lct_i and has for *processing time* p_i. During its execution, the task requires h_i units of a resource at each time point. Let a task i be represented by the tuple $\langle \mathrm{est}_i, \mathrm{lct}_i, p_i, h_i \rangle$. From these parameters, it is possible to compute the *earliest completion time* $\mathrm{ect}_i = \mathrm{est}_i + p_i$ and the *latest starting time* $\mathrm{lst}_i = \mathrm{lct}_i - p_i$ of a task i. The energy $e_i = p_i \cdot h_i$ represents the total amount of resource consumed during the execution of the task. The earliest starting time $\mathrm{est}_\Omega = \min_{i\in\Omega}(\mathrm{est}_i)$ of the set of tasks $\Omega \subseteq \mathcal{I}$ is the earliest time point at which any task in Ω can start. Similarly, the latest completion time $\mathrm{lct}_\Omega = \max_{i\in\Omega}(\mathrm{lct}_i)$ of Ω is the latest time point at which any task in Ω can end.

A task i has a *compulsory part* in the interval $[\mathrm{lst}_i, \mathrm{ect}_i)$ if $\mathrm{lst}_i < \mathrm{ect}_i$. The task must necessarily execute during its compulsory part since it cannot start later than its latest starting time and cannot end before its earliest completion time.

Using constraint programming, one can use the CUMULATIVE$(\vec{S}, \vec{p}, \vec{h}, C)$ constraint (Aggoun and Beldiceanu 1993) to enforce that, at any time point, the resource consumption of the tasks in execution is less than or equal to the capacity C of a resource. The constraint uses one starting time variable $S_i \in [\mathrm{est}_i, \mathrm{lst}_i]$ for each task i in the problem, as well as parameters for their processing times, heights, and the capacity of the resource.

Deciding whether the CUMULATIVE constraint can be satisfied is NP-Complete (Aggoun and Beldiceanu 1993). Filtering algorithms for the constraint can neither enforce domain or bounds consistency in polynomial time. Instead, it is necessary to rely on detection and filtering rules that work on a relaxation of the constraint. Many such rules have been introduced over the years. Rules relevant to this paper are presented below. Some rules have faster algorithms to apply them while others are slower but produce stronger inconsistency detection and filtering.

The Time-Tabling rule (1) (Beldiceanu and Carlsson 2002) uses a reasoning based on the compulsory parts of the

tasks. At any time point, if the sum of the height of the compulsory parts is greater than the capacity of the resource, the CUMULATIVE constraint cannot be satisfied. This checker rule is sufficient to enforce CUMULATIVE. We refer the reader to (Beldiceanu and Carlsson 2002) for the filtering rule based on the TimeTabling.

$$\exists t \sum_{i \in \mathcal{I} \,:\, \mathrm{lst}_i \leq t < \mathrm{ect}_i} h_i > C \implies \texttt{fail} \tag{1}$$

The EdgeFinding rule (2) (Mercier and Van Hentenryck 2008) checks for precedences between a set of tasks Ω and a task i. If the combined energy $e_{\Omega\cup\{i\}}$ of the tasks in Ω and i is greater than the energy available in the interval $[\mathrm{est}_{\Omega\cup\{i\}}, \mathrm{lct}_\Omega)$, then i must end after all the tasks in Ω have ended. Indeed, i is the only task that can execute outside of $[\mathrm{est}_{\Omega\cup\{i\}}, \mathrm{lct}_\Omega)$ and prevent an overflow in this interval.

$$\begin{aligned} &\forall i \in \mathcal{I}, \Omega \subseteq \mathcal{I} \setminus \{i\} \\ &e_{\Omega\cup\{i\}} > C \cdot (\mathrm{lct}_\Omega - \mathrm{est}_{\Omega\cup\{i\}}) \implies i \textbf{ ends after } \Omega \end{aligned} \tag{2}$$

Of particular interest for this paper is the energetic reasoning rule (Lopez and Esquirol 1996). It is one of the strongest rules, but also one of the slowest to apply. The energetic reasoning is based on the notion of left and right shift in an interval. Let $\texttt{LS}(i,l,u) = h_i \cdot (\min(u, \mathrm{ect}_i) - \max(l, \mathrm{est}_i))$ be the left-shift of task i in the interval $[l,u)$. It represents the amount of energy that the task consumes in the interval if it is scheduled at its earliest. The right-shift $\texttt{RS}(i,l,u) = h_i \cdot (\min(u, \mathrm{lct}_i) - \max(l, \mathrm{lst}_i))$ of task i in interval $[l,u)$ is symmetric and represents the amount of energy that task i consumes in the interval if it 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-shift and the right-shift. Regardless of when a task is scheduled, it always consumes at least $\texttt{MI}(i,l,u)$ units of energy in a given interval $[l,u)$, as shown on Figure 1.

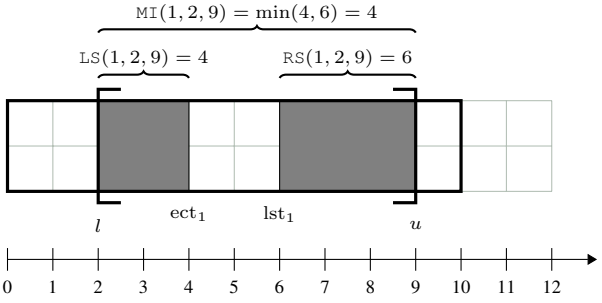

Figure 1: Example of the minimum intersection of task with $\mathrm{est}_1 = 0, \mathrm{lct}_1 = 10, p_1 = 4$ and $h_1 = 2$. The left gray area represents the left-shift of 4 while the right gray area represents the right shift of 6. The minimum intersection is the minimum between the left and right shift, which is 4. Regardless of where the task is scheduled, it always consume at least 4 units of energy in that interval.

Let $\texttt{MI}(\Omega,l,u) = \sum_{i\in\Omega} \texttt{MI}(i,l,u)$ be the sum of the minimum intersection of all the tasks in Ω. The satisfiability rule for the energetic reasoning (3) states that if there exists an interval for which this sum is greater than the energy available in that interval on the resource, then the constraint cannot be satisfied.

$$\exists l, \exists u,\, \texttt{MI}(\mathcal{I},l,u) > \mathrm{C} \cdot (u-l) \implies \texttt{fail} \tag{3}$$

Baptiste and al. (2001) showed that testing the rule on $O(n^2)$ intervals, called *intervals of interest*, is equivalent to testing the rule on all possible intervals. There are three types of intervals of interest (4). While there are $O(n^2)$ intervals of interest, there are also $O(n^2)$ distinct lower bounds and $O(n^2)$ distinct upper bounds. (Baptiste, Le Pape, and Nuijten 2001) proposed an $O(n^2)$ algorithm to apply the rule using the intervals of interest and (Ouellet and Quimper 2018) improved it to $O(n \log^2 n)$.

Intervals of interest

$$\begin{aligned} T_e = &\{[l,u) \mid l \in O_1, u \in O_2\} \cup \\ &\{[l,u) \mid l \in O_1, u \in O(l)\} \cup \\ &\{[l,u) \mid u \in O_2, l \in O(u)\} \end{aligned}$$

where

$$\begin{aligned} O_1 &= \{\mathrm{est}_i \mid i \in \mathcal{I}\} \cup \{\mathrm{ect}_i \mid i \in \mathcal{I}\} \cup \{\mathrm{lst}_i \mid i \in \mathcal{I}\} \\ O_2 &= \{\mathrm{lst}_i \mid i \in \mathcal{I}\} \cup \{\mathrm{ect}_i \mid i \in \mathcal{I}\} \cup \{\mathrm{lct}_i \mid i \in \mathcal{I}\} \\ O(t) &= \{\mathrm{est}_i + \mathrm{lct}_i - t\} \end{aligned} \tag{4}$$

It is also possible to filter the starting times using a rule similar to the checker rule. Baptiste proposed a $\Theta(n^3)$ algorithm to do so. Tesch (2018) and Ouellet and Quimper (2018) independently proposed algorithms in $O(n^2 \log n)$ and $O(n^2 \log^2 n)$ respectively.

Lazy clause generation

Several modern constraint programming solvers, such as Chuffed (Chu 2011) and OR-tools (Perron and Furnon 2019), use the lazy clause generation technique (Ohrimenko, Stuckey, and Codish 2009) to enhance their performances. The technique requires that the solver uses a SAT engine as an additional propagator. Each integer variable is also encoded with multiple Boolean variables, for the benefit of the SAT engine. For instance, an integer variable X can be encoded with boolean variables named $[\![X \geq 1]\!]$, $[\![X \geq 2]\!]$, etc. Furthermore, checker and filtering algorithms must explain the failures and the filtering they produced using SAT clauses. This allows the SAT engine to deduce *nogoods*, redundant SAT constraints that are added during the search to prevent the solver from making redundant unfruitful exploration. For instance, a checker algorithm that fails because the value of a variable X is too low could explain it with the clause $\neg[\![X \geq 5]\!] \Rightarrow \texttt{fail}$, where $\neg[\![X \geq 5]\!]$ is a literal negating the Boolean variable that encodes the fact that X is greater or equal to 5.

To allow the SAT engine to generate nogoods that can be reused more often, one produces the most general clause possible, with as few and as general literals as possible. For

instance, the literal $[\![X \geq 2]\!]$ is more general than $[\![X \geq 3]\!]$ because the latter implies the former.

It is often challenging to generate general explanations for global constraints, since, by their very nature, global constraints work on several variables, often leading to complex explanations. This means that, unlike constraint programming without nogoods, more filtering is not necessarily better, even without considering the running time of the algorithms. A decomposition of a global constraint that uses binary constraints with small and general explanations could be better than a global constraint with poor explanations but better filtering. Without nogoods, one would need to balance the running time of an algorithm with its filtering strength. With lazy clause generation, one must also consider the quality of the explanations. One can use experiments to find out the best balance between these three factors.

Related Work

De Clercq et al. (2010) introduced a constraint that extends the CUMULATIVE constraint with the following additional parameters and variables:

- A sequence $J = \langle 0, J_1, \ldots, J_{k-1}, \mathrm{lct}_{\mathcal{I}} \rangle$ forming consecutive intervals such that interval i is defined as $[J_{i-1}, J_i)$
- A sequence L of capacities such that $L_i \leq C$ is the capacity in the i-th interval of J.
- A sequence $\texttt{Cost}$ of integer variable such that $\texttt{Cost}_i$ is the overcost in the i-th interval of J.
- An integer variable Z representing the global penalty of the constraint.
- A function $\texttt{costFunction} \in \{\max, \mathrm{sum}\}$ indicating if the $\texttt{Cost}$ variables should be the maximum or the sum of the overcost in the intervals.
- A function $\texttt{penaltyFunction} \in \{\max, \mathrm{sum}\}$ indicating if the penalty variable Z should be the maximum between or the sum of the $\texttt{Cost}$ variables.

The authors proposed both a $O((n + |J|) \cdot \log(n + |J|))$ TimeTabling and an $O(n \cdot |J| \cdot k \cdot \log(n))$ EdgeFinding algorithm, where $k \leq n$ is the number of distinct heights among the tasks. They do not generate explanations of the filtering.

3 SoftCumulative

We define the SOFTCUMULATIVE constraint as follows.

$$\begin{aligned} &\textsc{SoftCumulative}(\vec{S}, \vec{p}, \vec{h}, C, Z, f) \iff \\ &Z \geq \sum_t f\big(\max(0, \sum_{i \in \mathcal{I}: S_i \leq t < S_i + p_i} h_i - C\big) \end{aligned} \tag{5}$$

The variable $S_i \in [\mathrm{est}_i, \mathrm{lst}_i]$ is the starting time of task i. The parameters $\vec{p}$, $\vec{h}$, and C represent respectively the processing times, the heights, and the capacity of the resource. The variable Z is the overcost variable representing the penalty incurred for *overflowing* the capacity of the resource. The function $f(x)$ is a non-decreasing function returning the penalty for overflowing the capacity by x units at one time point. We use the terms *penalty* and *cost* interchangeably. We consider two cost functions: a linear cost $f(x) = x$ and a quadratic cost $f(x) = x^2$. A quadratic cost function is interesting in cases where spreading the overflow over multiple time points is preferable to having few time points with high overflow. For instance, if one unit of the resource represents one employee, it is often more cost-effective to hire one additional employee for a few days than hire multiple additional employees for only one day. With a quadratic cost function and a capacity of 0, SOFTCUMULATIVE can be used to spread the tasks as equally as possible along the time line. Although we focus on the linear and quadratic cost functions, the algorithms we present can be easily adapted to use other non-decreasing cost functions.

The CUMULATIVE constraint is a specific case of the SOFTCUMULATIVE constraint where $Z = 0$. This means that it is also NP-Complete to decide whether SOFTCUMULATIVE has a solution. This also means that SOFTCUMULATIVE is at least as hard as CUMULATIVE and that filtering algorithms for SOFTCUMULATIVE cannot be faster than their equivalent for the CUMULATIVE.

We extend the energetic reasoning to support the cost function $f(x)$. Let $\texttt{overcost}(l, u)$ be the overcost of an interval $[l, u)$. If the amount of energy available in that interval is greater than or equal to the energy consumed by the tasks, the overcost is zero. Otherwise, the overcost is computed as follows.

$$\begin{aligned} \texttt{overcost}(l,u) =& f(\kappa) \cdot (u-l) + \\ & (f(\kappa+1) - f(\kappa)) \cdot (S \bmod (u-l)) \\ \textbf{where } S =& \texttt{MI}(\mathcal{I}, l, u) - \mathrm{C} \cdot (u-l), \\ \kappa =& \left\lfloor \frac{S}{u-l} \right\rfloor \end{aligned} \tag{6}$$

Note that with a linear cost function, the overcost is simply the amount of energy that overflows from the interval.

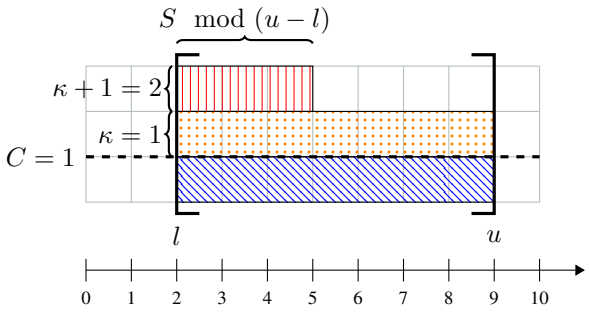

Figure 2: Overcost computation

Figure 2 presents an example of how the overcost in the interval $[2, 9)$, with a resource of capacity $C = 1$, and a minimum intersection $\texttt{MI}(\mathcal{I}, 2, 9) = 17$ is calculated. 7 units of energy available on the resource are represented with diagonal blue lines. 7 units of energy where the capacity of the resource is exceeded by 1 unit is represented with orange dotted points. That leaves 3 units of energy for which the capacity of the resource is exceeded by 2 units. These units

span only a part of the interval. They are represented with red vertical lines. For a linear penalty $f(x) = x$, we obtain $\texttt{overcost}(2, 9) = 10$ while a quadratic penalty $f(x) = x^2$ gives $\texttt{overcost}(2, 9) = 16$.

4 Checker algorithm

We propose a checker algorithm for SOFTCUMULATIVE based on the energetic reasoning. With the CUMULATIVE constraint, it is sufficient to find one interval with a minimum intersection greater than the available energy to detect an inconsistency. With SOFTCUMULATIVE, the checker algorithm instead needs to compute a lower bound on the overcost variable. If it is greater than the upper bound, there is a failure. The approach of the classic energetic checker of finding one interval where there is an overload can work for computing a lower bound. However, this approach considers only one interval at a time. Considering multiple intervals and combining their overcost lead to a better bound. Hence, a better solution is to partition the time line into disjoint intervals such that the sum of the overcost is maximized.

We propose to solve this problem using a graph G (see Figure 3). There is one node for each time point. For each pair of time points l and u such that $l < u$, there is an arc (l, u) with weight $\texttt{overcost}(l, u)$. Each arc (l, u) corresponds to an semi-open interval $[l, u)$ of positive length. The goal is to find the longest path between the first time point and the last. The sum of the weights of the arcs on that longest path gives us a lower bound $\underline{Z}$ on the overcost variable Z. Since the graph is acyclic, we can solve this problem in polynomial time using dynamic programming.

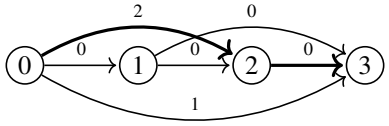

Figure 3: Example of a graph representing the overcost for a SOFTCUMULATIVE with a capacity of $C = 1$ with 4 tasks of the form $\langle 0, 2, 1, 1\rangle$. There is $C \cdot (u - l) = 1 \cdot 2 = 2$ units of energy available on the resource in $[0, 2)$. Each task has a minimum intersection of 1 in that interval. This means there is $4 - 2 = 2$ units of overcost in the interval $[0, 2)$, making it part of the the longest path (which is in bold).

This proposed rule adapts the energetic reasoning to the SOFTCUMULATIVE constraint and Algorithms 1 & 2 apply this rule.

$$\exists \langle P_0, \ldots, P_k\rangle, P_0 = 0 \wedge P_k = \mathrm{lct}_{\mathcal{I}} \wedge P_t < P_{t+1} \wedge \sum_{t \in \{0..k-1\}} \texttt{overcost}(P_t, P_{t+1}) > \underline{Z} \implies \texttt{fail} \quad (7)$$

Algorithm 1 takes as input the set of tasks, the capacity of the resource, and a set of critical time points T. By changing the number of critical time points, we can change the complexity of the algorithm, which is $\Theta(|T|^2)$, but also the quality of the resulting lower bound. With more time points, the algorithm is slower, but the lower bound is better. The `OvercostBound` algorithm returns the lower bound and the vector π, a parent vector containing the choices made by the dynamic programming. The vector π is not relevant to the checker, but it is later used by the filtering algorithm. Once the lower bound is computed, we can check whether it is greater than the upper bound $\overline{Z}$ of the overcost (Algorithm 2). If so, the algorithm returns a failure.

Algorithm 1: OvercostBound($\mathcal{I}, \mathrm{C}, \mathrm{T}$)

```
Φ ← 0⃗;
π ← [nil, ..., nil];
for u = 2..|T| do
    for l = 1..u − 1 do
        c ← Φ[l] + overcost(T[l], T[u]));
        if Φ[u] < c then
            Φ[u] ← c;
            π[u] ← l;
return ⟨Φ[|T|], π⟩;
```

Algorithm 2: SoftEnergeticChecker($\mathcal{I}, \mathrm{C}, \mathrm{T}, Z$)

```
⟨α, π⟩ ← OvercostBound(I, C, T);
if α ≤ Z̄ then return pass else return fail;
```

Let T_e be the set of $O(n^2)$ time points in (4) used by the intervals of interest of Baptiste et al. (2001). Let $T_s = \{\mathrm{est}_i \mid i \in \mathcal{I}\} \cup \{\mathrm{ect}_i \mid i \in \mathcal{I}\} \cup \{\mathrm{lst}_i \mid i \in \mathcal{I}\} \cup \{\mathrm{lct}_i \mid i \in \mathcal{I}\}$ be the set of $4n$ *critical time points* of each task.

Theorem 1. *If Z is set to 0 and T to T_e, then `SoftEnergeticChecker` is equivalent to the energetic checker for the* CUMULATIVE *constraint.*

Proof. With an upper bound on 0 for the overcost, `SoftEnergeticChecker` returns a failure if and only if the computed lower bound is 1 or more. There are two cases.

If the classic energetic checker passes, there does not exist an interval for which the capacity is exceeded. This means that the overcost of all intervals is 0. In that case, the lower bound found by our algorithm is 0 and the `SoftEnergeticChecker` passes.

If the classic energetic checker fails, there is at least one interval for which the capacity is exceeded (and for which the overcost is greater than 0) and that interval is one of the intervals of interest. While searching for the longest path, the `SoftEnergeticChecker` processes all possible intervals of positive length formed by the time points in T. Since T corresponds to the lower bounds and upper bounds of the interval of interest, all intervals of interest (and some intervals that are not of interest) are examined, including the one with a positive overcost. Hence, the longest path has a cost of at least 1 and the `SoftEnergeticChecker` fails. □

Since there are $O(n^2)$ lower bounds and $O(n^2)$ up-

per bounds in T_e, considering them all would lead to a `SoftEnergeticChecker` with a complexity of $O(n^4)$, which is not reasonable for an algorithm that is executed thousands of times during the search. Instead, we propose to use the subset T_s that is linear in size. This subset gave us good results while keeping the complexity of the algorithm reasonable. Furthermore, this subset is sufficient to apply two weaker rules, the Time-Tabling and the Edge-Finding, as shown in Theorem 2.

Theorem 2. *If $Z = 0$ and the set $\mathrm{T} = T_s$, then the `SoftEnergeticChecker` applies the Time-Tabling.*

Proof. The Time-Tabling rule detects a failure if, at any time point, the sum of the compulsory parts of the tasks is greater than the capacity of the resource. Suppose that t is such a time point.

Let lst_i be the latest lst less than or equal to t and let ect_j be the earliest ect greater than or equal to t. By definition, the total amount of energy consumed by the compulsory parts can only increase at time points that are lst and it can only decrease at time points that are ect. Thus, the amount of energy consumed by the compulsory parts at each time point in $[\mathrm{lst}_i, \mathrm{ect}_j)$ is the same as the amount at t. Since the amount consumed at t is sufficient to overload the resource, the total amount of energy in $[\mathrm{lst}_i, \mathrm{ect}_j)$ is sufficient to overload the interval.

Our checker algorithm examines every interval formed from time points in T and all ect and lst are in that set. Thus, our checker raises a failure when examining $[\mathrm{lst}_i, \mathrm{ect}_j)$. □

5 Filtering algorithm

We propose an algorithm based on the checker for filtering the earliest starting time of the tasks (see Algorithm 3). Filtering the latest completion time is symmetric.

The idea behind the algorithm is to schedule a task to its earliest and then run the checker algorithm to check if it is a valid solution. If it is not, we can filter the earliest starting time. We use two strategies to save costly calls to the checker. First, we use a constant time check to see whether it is possible that scheduling a task to its earliest makes the checker fail. Second, we use the longest path returned by the checker in case of a failure to filter the earliest starting time of the task as much as possible without recalling the checker. We do this by sliding the execution of the task from its earliest starting time up to a point where the cost of the path no longer causes a failure. Each time the task is shifted by one unit of time, the weight of at most four edges on the path needs to be updated, namely the edges that span the intervals that contain est_i, $\mathrm{est}_i + 1$, ect_i, and $\mathrm{ect}_i + 1$.

The algorithm starts by computing a lower bound $\underline{Z}$ on the overcost using the checker. Then, for each task i, it performs a check, at line 1, to verify whether the task needs to be filtered. The check changes slightly depending on the cost function. In the linear case, if the free energy of the task, which is the portion of the task that is not compulsory, is greater than the difference between the computed lower bound and the upper bound on the overcost, the task *might* need to be filtered. Otherwise, scheduling the task at its earliest cannot sufficiently increase the lower bound to cause a failure and the task does not need to be filtered. Indeed, by fixing task i to its earliest, we are effectively only using its left-shift. Thus, the minimum intersection in some intervals can increase, but only by the free energy. The energy from the compulsory part is already included because, by definition, it is both in the left and right shift. This check gains in importance as the search reaches the bottom of the search tree. More tasks are fixed, fewer tasks has free energy, and fewer tasks need to be filtered. For a non-linear cost function, one additional unit of energy can increase the cost by more than one unit. Thus, the check only verifies whether the task has free energy.

If the algorithm finds that a task needs to be filtered, it fixes the task to its earliest starting time and runs the checker. If there is a failure, the task cannot be executed at its earliest and needs to be filtered. To do so, the algorithm increment its est_i by one. To continue the filtering process, lct_i is also incremented. The process is repeated until the computed lower bound on the overcost does not exceed the upper bound.

Algorithm 3: SoftEnergeticFiltering($\mathcal{I}, \mathrm{C}, \mathrm{T}, \overline{Z}, f$)

```
eMin ← 0;
if f is a linear function then
    Z ← OvercostBound(I, C, T);
    eMin ← Z̄ − Z;
for i ∈ I do
1   if h_i · (p_i − max(0, ect_i − lst_i)) > eMin then
        t ← lct_i;
        lct_i ← est_i + p_i;
        Z' ← OvercostBound(I, C, T);
        while Z' > Z̄ do
            est_i ← est_i + 1;
            lct_i ← lct_i + 1;
2           Z' ← OvercostBound(I, C, T);
        lct_i ← t;
return {est_i | i ∈ I};
```

Algorithm 4: ComputeLongestPath($\mathcal{I}, \mathrm{C}, \mathrm{T}, \pi$)

```
u ← |T|;
P ← [T[u]];
while u > 1 do
    u ← π[u];
    Push T[u] to the front of P;
return P;
```

Note that even if the checker algorithm initially reports a success, it is possible for the filtering algorithm to remove all possible values for the starting time of a task, leading to a failure. This is because the energetic reasoning relaxation allows all tasks to be preempted. However, by fixing a task to its earliest, our filtering algorithm removes the possibility of preemption for the task being filtered.

Algorithm 5: AdjustCost($\mathcal{I}, i, P$)

Create a task i' with $\langle \text{est}_i + 1, \text{lct}_i + 1, p_i, h_i \rangle$;
$\mathcal{I}' \leftarrow \mathcal{I} \setminus \{i\} \cup \{i'\}$;
$D \leftarrow \{k \mid \exists x \in \{\text{est}_i, \text{est}_i + 1, \text{ect}_i, \text{ect}_i + 1\}, P_k \leq x < P_{k+1}\}$;
return $\sum_{k \in D} \texttt{overcost}(\mathcal{I}', P_k, P_{k+1}) - \sum_{k \in D} \texttt{overcost}(\mathcal{I}, P_k, P_{k+1})$;

To improve the efficiency of the algorithm, one can compute the longest path P using `ComputeLongestPath`, then replace line 2 by $\underline{Z}' \leftarrow \underline{Z}' + \texttt{AdjustCost}(\mathcal{I}, i, P)$. This allows filtering task i as much as possible on the longest path without having to re-run the checker. The lower bound $\underline{Z}'$ is instead adjusted incrementally. Algorithm 5 works as follows. Since the increment is only by one, the minimum intersection can change in at most four intervals: the interval containing the current est_i, the interval containing the current ect_i and the intervals immediately following these two intervals if the est_i or ect_i is the upper bound on the interval. Computing the change in cost is done in constant time by keeping in cache the minimum intersection for each interval.

With this adjustment, the complexity of the algorithm is $O(k \cdot |\,\mathrm{T}\,|^2 + r)$ where $k \leq n$ is the number of tasks for which the free energy check passes and r is the total number of values removed from the domains. Even if r can be as high as the makespan, it is not necessarily bad since each value removed from the domain of a variable helps reduce the search space. We want to minimize the amount of time spent by the algorithm that leads to no filtering. Spending time if we are sure to filter is acceptable. If we use the set T_s of $4n$ time points as we suggest for the checker algorithm, we have a complexity of $O(k \cdot n^2 + r)$. If we instead use the set T_e, we have a complexity of $O(k \cdot n^4 + r)$.

Note that it is also possible to increase the earliest starting time by more than one unit at a time, but such an adjustment depends on the cost function $f(x)$.

Theorem 3. *If $Z = 0$ and the set $T = T_s$ the `SoftEnergeticFiltering` algorithm applies the EdgeFinding rule.*

Proof. Suppose the EdgeFinding rule (2) finds a precedence stating that a set Ω must end before task i ends. The `SoftEnergeticFiltering` algorithm filters task i until the precedence is no longer detected.

Indeed, if the EdgeFinding rule holds for i and Ω, this means that there is not enough energy in $[\text{est}_{\Omega \cup \{i\}}, \text{lct}_\Omega)$ for both the energy of the tasks in Ω and the energy of i, given that i is left-shifted. In other words, when i is left-shifted, $\texttt{overcost}(\text{est}_{\Omega \cup \{i\}}, \text{lct}_\Omega) > 0$. Since both the lower bound and the upper bound on that interval are est and lct time points, they are in T_s. Thus, when filtering task i, the `SoftEnergeticFiltering` algorithm finds at least one path with positive overcost. Since $Z = 0$, task i is filtered by at least 1 unit. This reasoning holds as long as the precedence is detected by the EdgeFinding rule. □

6 Explanations

We show how to generate explanations to allow our algorithms to be used with lazy clause generation solvers. The goal is to have explanations with as few and as general literals as possible. However, due to the global nature of SOFTCUMULATIVE, failures or filtering is caused by the interaction between several tasks and the overcost variable. This means that our explanations will necessarily contain multiple variables and will not be as general as explanations from binary constraints.

To explain a failure, we use an explanation of the form $\texttt{conditions} \implies \texttt{fail}$. To explain filtering, we instead use the form $\texttt{conditions} \implies [\![S_i \geq \text{est}'_i]\!]$. The conditions are the same for failure and for filtering since it is caused, in both cases, by an excess in the energy of some tasks in the intervals along a path. For that reason, the rest of the section focuses on the conditions in the left side of the implication.

In any explanation for SOFTCUMULATIVE, we must include the literal $[\![Z \leq \overline{Z}]\!]$ for the upper bound on the overcost variable.

There are several ways to explain the start variables of the tasks. A simple, naive explanation would include the literals $[\![S_i \geq \text{est}_i]\!] \wedge [\![S_i \leq \text{lst}_i]\!]$ for each task i. However, not all the tasks in the scope of SOFTCUMULATIVE contribute to the failure or filtering. In fact, only the tasks that have a positive minimum intersection in one of the intervals with positive overcost on the longest path contribute to the failure/filtering. This means we can exclude the other tasks from the explanation.

It is possible to generalize some literals. The minimum intersection of a task in an interval comes from the minimum between the left-shift and the right-shift. We can reduce the maximum between the left and the right shift up to the other value. For instance, if the left-shift of a task is 4 and its right-shift is 3, we can reduce the left-shift to 3 while still keeping our explanation valid. We can do that by increasing the est for the left-shift and reducing the lst for the right-shift, as shown in (8). Note that this rule may produce multiple literals with the same variable. It is possible to simplify the literals $[\![S_i \geq a]\!] \wedge [\![S_i \geq b]\!]$ to $[\![S_i \geq \max(a, b)]\!]$ to keep only the most restrictive literal.

$$[\![Z \leq \underline{Z}]\!] \wedge \bigwedge_{\substack{i \in \mathcal{I}, 1 \leq k < |P|: \\ \texttt{overcost}(P_k, P_{k+1}) > 0 \wedge \texttt{MI}(i, P_k, P_{k+1}) > 0}} [\![S_i \geq P_{k+1} - x]\!] \wedge [\![S_i \leq P_k + x + p_i]\!]$$
$$\textbf{where } x = \frac{\min(\texttt{LS}(i, l, u), \texttt{RS}(i, l, u))}{h_i} \tag{8}$$

7 Experiments

To ascertain the practical relevance of our SOFTCUMULATIVE checker and filtering algorithms, we used an adaptation to the Resource Constrained Project Scheduling Problem (RCPSP), which is a problem often used in the literature to benchmark the CUMULATIVE constraint.

In the RCPSP problem, the goal is to schedule n tasks on m machines of various capacities. Each task has a fixed

processing time and is executed simultaneously on all machines. The amount of resource a task consumes varies by machine and can be zero. There are precedence constraints between tasks and the goal is to minimize the makespan.

With SOFTCUMULATIVE, we instead want to minimize the cost of overloading the resources. It is possible to adapt existing RCPSP instances (that do not allow overloads) by finding the makespan of the optimal solution and then either reducing the capacity of the resources or decreasing the makespan. With this approach, the makespan becomes a parameter of the instance and the goal is now to minimize the sum of overcost over all resources. We tried both approaches, but, with our benchmarks, reducing the capacity of the resources gave us more interesting instances. Reducing the makespan often produced unsatisfiable instances due to the precedence constraints. We used an adapted version of the KSD15_D benchmark (Koné et al. 2011). These instances are highly cumulative (more than one task can often execute at a time). We adapted it by decreasing the capacity of each resource by four units and fixing the makespan to the optimal value of the original instance. This gave us interesting instances, several of which can be solved in a reasonable time. These adapted instances are available in the supplementary material.

We implemented our algorithms in C++ and used the lazy clause generation solver Chuffed (Chu 2011). We used the modelling language Minizinc (Nethercote et al. 2007) to implement our models. Our experiments were run on an Intel Xeon 4110 CPU with a speed of 2.10Ghz. We ran all our experiments with a timeout of 20 minutes.

In our experiments, we compared our algorithms to the decomposition of SOFTCUMULATIVE (see Equation 5). We tested two configurations of our algorithms: the checker alone and the checker combined with the filtering algorithm. For both algorithms, we set $T = T_s$. In our experiments, the checker and filtering combination always outperformed the checker alone, so we only report the former to save space. The interested reader can find the raw results in the annex, including the checker alone configuration.

(De Clercq et al. 2010) kindly gave us access to their Java code that uses the solver Choco 2. We tried to solve our instances with their constraint. However, they only implemented the version of the constraint that computes the sum of the maximum overloads in each interval. To model our problem with this version, we need one interval per time point. Hence, the complexity depends on the makespan. This is not a good match for our instances with hundreds of time points. Even for easy instances, we were not able to find the optimal solution within a few hours with this approach. In addition, their constraint does not support a quadratic cost function, which is the most interesting application of our SOFTCUMULATIVE. Furthermore, naive explanations must be used with their algorithms since no method of generating explanations was proposed for these algorithms. Conversely, our constraint cannot be used to solve their instances since we do not support a maximum penalty function.

Linear cost function

Figure 4 (left) compares the time taken by the decomposition and the filtering algorithm when a linear cost function $f(x) = x$ is used. Almost all instances are solved much more quickly with the SOFTCUMULATIVE constraint and its filtering algorithm than by the decomposition. Furthermore, there are many instances for which the decomposition times out at 20 minutes (the points at 1200 seconds) while the filtering algorithm is able to solve them to optimality.

The decomposition is better on only 1% of the instances. Both configurations failed to solve 21.5% of the instances to optimality and the filtering algorithm is better on 77.5% of the instances. Furthermore, for nearly 68% of the instances, the filtering algorithm is more than 10 times faster.

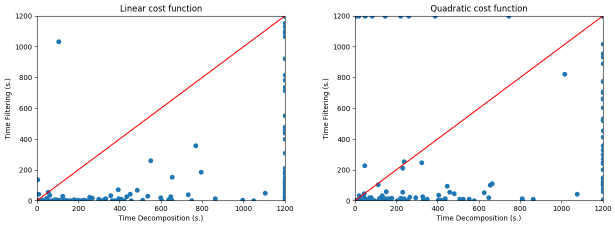

Figure 4: Time in seconds to solve the instances of the adjusted KSD15_D benchmark to optimality with a linear (left) and quadratic (right) cost function for the decomposition and the filtering algorithm. A time of 1200s means that the solver times out without proving the optimal solution.

Quadratic cost function

Figure 4 compares the time taken to solve the instances of the KSD15_D benchmark with the quadratic cost function. The results are similar to the linear case. Neither algorithm proved optimality for 24.6% of the instances. The decomposition is better for 2.7% of the instances while the filtering algorithm is better on 72.7% of the instances. The filtering algorithm is 10 times faster for 60% of the instances. Hence, the filtering algorithm is the method of choice to employ with both linear and quadratic cost functions.

8 Conclusion

We proposed a checker and a filtering algorithm based on the energetic reasoning for the SOFTCUMULATIVE constraint. Unlike previous work, both algorithms support a quadratic cost function in addition to a linear function. They are parametrable in the sense that their strength and complexity vary depending of the number of time points passed as parameters. With the recommended set of time points T_s, the checker has a complexity of $O(n^2)$ and the filtering algorithm has a complexity of $O(k \cdot n^2 + r)$ where k is the number of tasks that pass the free energy check and r is the number of values pruned from the domain of the variables after the execution of the algorithm. We showed how to explain the algorithms to use them with lazy clause generation solvers and presented evidence that, in practice, our algorithms vastly outperformed the decomposition.

References

Aggoun, A.; and Beldiceanu, N. 1993. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and computer modelling*, 17(7): 57–73.

Baptiste, P.; Le Pape, C.; and Nuijten, W. 2001. *Constraint-Based Scheduling*. Kluwer Academic Publishers.

Beldiceanu, N.; and Carlsson, M. 2002. A new multi-resource cumulatives constraint with negative heights. In *International Conference on Principles and Practice of Constraint Programming*, 63–79. Springer.

Chu, G. 2011. *Improving combinatorial optimization*. Ph.D. thesis, Department of Computing and Information Systems, University of Melbourne.

De Clercq, A.; Petit, T.; Beldiceanu, N.; and Jussien, N. 2010. A soft constraint for cumulative problems with overloads of resource. In *Doctoral Programme of the 16th International Conference on Principles and Practice of Constraint Programming (CP'10)*, 49–54.

Koné, O.; Artigues, C.; Lopez, P.; and Mongeau, M. 2011. Event-based MILP models for resource-constrained project scheduling problems. *Computers & Operations Research*, 38(1): 3–13.

Lopez, P.; and Esquirol, P. 1996. Consistency enforcing in scheduling: A general formulation based on energetic reasoning. In *5th International Workshop on Project Management and Scheduling (PMS'96)*.

Mercier, L.; and Van Hentenryck, P. 2008. Edge finding for cumulative scheduling. *INFORMS Journal on Computing*, 20(1): 143–153.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. MiniZinc: Towards a standard CP modelling language. In *International Conference on Principles and Practice of Constraint Programming*, 529–543. Springer.

Ohrimenko, O.; Stuckey, P. J.; and Codish, M. 2009. Propagation via lazy clause generation. *Constraints*, 14(3): 357–391.

Ouellet, Y.; and Quimper, C.-G. 2018. A $O(n^2 \log n)$ 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*, 477–494. Springer.

Perron, L.; and Furnon, V. 2019. OR-Tools.

Tesch, A. 2018. Improving energetic propagations for cumulative scheduling. In *International conference on principles and practice of constraint programming*, 629–645. Springer.