Generalizing the Edge-Finder Rule for the Cumulative Constraint

Vincent Gingras
Université Laval, Québec, QC, Canada
vincent.gingras.5@ulaval.ca

Claude-Guy Quimper
Université Laval, Québec, QC, Canada
Claude-Guy.Quimper@ift.ulaval.ca

Abstract

We present two novel filtering algorithms for the CUMULATIVE constraint based on a new energetic relaxation. We introduce a generalization of the Overload Check and Edge-Finder rules based on a function computing the earliest completion time for a set of tasks. Depending on the relaxation used to compute this function, one obtains different levels of filtering. We present two algorithms that enforce these rules. The algorithms utilize a novel data structure that we call *Profile* and that encodes the resource utilization over time. Experiments show that these algorithms are competitive with the state-of-the-art algorithms, by doing a greater filtering and having a faster runtime.

1 Introduction

Scheduling consists of deciding when a set of tasks needs to be executed on a shared resource. Applications can be found in economics [Buyya *et al.*, 2005] or in industrial sequencing [Harjunkoski *et al.*, 2014].

Constraint programming is an efficient way to solve scheduling problems. Many powerful filtering algorithms that prune the search space have been introduced for various scheduling problems [Baptiste *et al.*, 2001]. These algorithms are particularly adapted for the cumulative problem in which multiple tasks can be simultaneously executed on a cumulative resource. Among these algorithms, we note the Time-Table [Beldiceanu and Carlsson, 2002], the Energetic Reasoning [Lopez and Esquirol, 1996], the Overload Check [Wolf and Schrader, 2006], the Edge-Finder [Mercier and Van Hentenryck, 2008] and the Time-Table Edge-Finder [Vilím, 2011].

Constraint solvers call filtering algorithms multiple times during the search, hence the need for them to be fast and efficient. Cumulative scheduling problems being NP-Hard, these algorithms rely on a relaxation of the problem in order to be executed in polynomial time. In this paper, we introduce a novel relaxation that grants a stronger filtering when applied in conjunction with known filtering algorithms.

In the next section, we formally define what a Cumulative Scheduling Problem (CuSP) is. Then, we present two state-of-the-art filtering rules: the Overload Check and Edge-Finder. We generalize these rules so that they become function of the earliest completion time of a set of tasks. We introduce a novel function computing an optimistic value of earliest completion time for a set of tasks, based on a more realistic relaxation of the CuSP. Along with this function, we present a novel data structure, named *Profile*, we use to compute the function. We introduce two algorithms to enforce the generalized rules while using our own novel function. Finally, we present experimental results obtained while solving CuSP instances from two different benchmark suites.

2 The Cumulative Scheduling Problem

We consider the scheduling problem where a given set of tasks $\mathcal{I} = \{1, \ldots, n\}$ must be executed, without interruption, on a cumulative resource of capacity C. A task $i \in \mathcal{I}$ has an earliest starting time $\mathrm{est}_i \in \mathbb{Z}$, a latest completion time $\mathrm{lct}_i \in \mathbb{Z}$, a processing time $p_i \in \mathbb{Z}^+$, and a resource consumption value, commonly referred as height, $h_i \in \mathbb{Z}^+$. The energy of a task i is given by $e_i = p_i h_i$. We denote the earliest completion time of a task $\mathrm{ect}_i = \mathrm{est}_i + p_i$ and the latest starting time $\mathrm{lst}_i = \mathrm{lct}_i - p_i$. Some of these parameters can be generalized for a set of tasks $\Omega \subseteq \mathcal{I}$.

$$\mathrm{est}_\Omega = \min_{i \in \Omega} \mathrm{est}_i \qquad \mathrm{lct}_\Omega = \max_{i \in \Omega} \mathrm{lct}_i \qquad e_\Omega = \sum_{i \in \Omega} e_i$$

Let S_i be the starting time of task i, and its domain be $\mathrm{dom}(S_i) = [\mathrm{est}_i, \mathrm{lst}_i]$. The constraint CUMULATIVE$([S_1, \ldots, S_n], C)$ is satisfied if the total resource consumption of the tasks executing at any time t does not exceed the resource capacity C, which is expressed as :

$$\forall t : \sum_{i \in \mathcal{I}, S_i \leq t < S_i + p_i} h_i \leq C. \tag{1}$$

A solution to the CUMULATIVE constraint is a solution to the Cumulative Scheduling Problem (CuSP). In addition to satisfying the CUMULATIVE constraint, one usually aims at optimizing an objective function, such as minimizing the makespan, i.e. the time at which all tasks are completed.

Such scheduling problems are NP-Hard [Garey and Johnson, 1979], therefore it is NP-Hard to remove every inconsistent values from the domains of the starting time variables S_i. However, there exist many powerful filtering algorithms running in polynomial time for the CUMULATIVE constraint. To

execute in polynomial time, these algorithms rely on a relaxation of the original problem that generally revolves around a task property referred as the elasticity [Baptiste *et al.*, 2001]. A task i becomes fully elastic if we allow its resource consumption to fluctuate (and even to interrupt), as long as the amount of resource consumed in the interval $[\mathrm{est}_i, \mathrm{lct}_i)$ is equal to its energy e_i.

3 Preliminaries

We present two filtering algorithms based on an energetic relaxation that we later improve using a novel relaxation.

3.1 Overload Check

The Overload Check is a test that detects inconsistencies in the problem and triggers backtracks in the search tree. The Overload Check rule enforces the condition that the energy consumption required by a set of tasks Ω cannot exceed the resource capacity over the time interval $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega)$.

$$\exists \Omega \subseteq \mathcal{I} : C(\mathrm{lct}_\Omega - \mathrm{est}_\Omega) < e_\Omega \Rightarrow \text{fail} \tag{2}$$

This condition is necessary to the existence of a feasible solution to the problem. [Wolf and Schrader, 2006] present an algorithm enforcing this rule running in $O(n \log n)$ time. More recently, [Fahimi and Quimper, 2014] presented an Overload Check algorithm that runs in $O(n)$ time, using a data structure named *Timeline*. Although initially conceived for the DISJUNCTIVE constraint, it is demonstrated that the algorihtm can be adapted for the CUMULATIVE constraint, while maintaining its running time complexity of $O(n)$.

3.2 Edge-Finder

The Edge-Finder algorithm filters the starting time variables. The algorithms by [Vilím, 2009] and [Kameugne *et al.*, 2014] proceed in two phases : the detection and the adjustment.

The detection phase detects *end before end* temporal relations between the tasks. The relation $\Omega \ll i$ indicates that the task i must finish after all tasks in Ω are completed.

Given a set of tasks $\Omega \subseteq \mathcal{I}$, the Edge-Finder detection rule enforces the condition that if a task $i \notin \Omega$ cannot be concurrently executed along the tasks in Ω without having any of them missing their deadline, then the tasks in Ω must end before the task i ends, i.e. $\Omega \ll i$. [Baptiste *et al.*, 2001] present the following rule.

$$e_{\Omega \cup \{i\}} > C(\mathrm{lct}_\Omega - \mathrm{est}_{\Omega \cup \{i\}}) \Rightarrow \Omega \ll i \tag{3}$$

[Vilím, 2009] detects all precedences using this rule. He demonstrates that the rule does not require to be applied for all subsets. Let the left cut of task i be the set of tasks whose lct is no greater than lct_i, i.e. $\mathrm{LCut}(\mathcal{I}, i) = \{j \in \mathcal{I} \mid \mathrm{lct}_j \leq \mathrm{lct}_i\}$. Then the algorithm only needs to enforce the rule (3) for all distinct left cuts. Additionally, Vilím's detection algorithm introduces the Θ-Λ-tree data structure to achieve a $O(n \log n)$ time complexity.

Assuming a precedence relation $\Omega \ll i$ is found during the detection phase, the adjustment phase proceeds to filter the earliest starting time of task i. The new value is computed by spending the energy e_Ω in the time interval $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega)$ as follows. A maximum amount of energy is spent on the interval with a restricted capacity of $C - h_i$. The remaining energy, i.e. $\max(0, e_\Omega - (\mathrm{lct}_\Omega - \mathrm{est}_\Omega)(C - h_i))$, is spent as early as possible on the interval using the remaining capacity h_i. The time when this remaining energy completes its execution is the new bound est_i. [Baptiste *et al.*, 2001] present the following rule for the adjustment phase.

$$\Omega \ll i \Rightarrow \mathrm{est}_i \geq \max_{\Omega' \subseteq \Omega} \left\{ \mathrm{est}_{\Omega'} + \left\lceil \frac{e_{\Omega'} - (C - h_i)(\mathrm{lct}_{\Omega'} - \mathrm{est}_{\Omega'})}{h_i} \right\rceil \right\} \tag{4}$$

The main difficulty of the adjustment phase is to compute the subset Ω' that results in the optimal adjustment. [Vilím, 2009] introduces an adjustment algorithm running in $O(kn \log n)$ time, where k is the number of distinct heights. His algorithm uses an extended Θ-Λ-tree. In an orthogonal work, [Kameugne *et al.*, 2014] introduce an adjustment algorithm running in $O(n^2)$ time, based on notions of minimum slack and maximum density. Although their algorithm does not strictly dominate Vilím's algorithm complexity, they demonstrate that it performs better in practice.

4 Novel function of earliest completion time

The earliest completion time of a set of tasks Ω, denoted ect_Ω, is NP-Hard to compute [Garey and Johnson, 1979]. One normally uses a relaxation in order to compute an optimistic (smaller) value of earliest completion time. This relaxation can be identified as the fully-elastic relaxation [Baptiste *et al.*, 2001]. In order to differentiate the two functions of ect presented in this paper, we took the discretion to rename the function of ect presented by [Vilím, 2009] as $\mathrm{ect}^{\mathrm{F}}$. It is computed by spending a maximum amount of energy as early as possible without any regards to the heights of the tasks. [Vilím, 2009] uses the following formula to compute it.

$$\mathrm{ect}^{\mathrm{F}}_\Omega = \left\lceil \frac{\max\{C\,\mathrm{est}_{\Omega'} + e_{\Omega'} \mid \Omega' \subseteq \Omega\}}{C} \right\rceil \tag{5}$$

We introduce a generalization of the Overload Check rule based on the function ect or any of its relaxation. If a particular set of tasks cannot be completed before the deadline of this set, then the problem does not have a solution.

$$\exists \Omega \subseteq \mathcal{I} : \mathrm{ect}_\Omega > \mathrm{lct}_\Omega \implies \text{fail} \tag{6}$$

Substituting the function ect by the fully-elastic relaxed $\mathrm{ect}^{\mathrm{F}}$ (5) into rule (6) leads to the well known form of the Overload Check rule (2). The demonstration follows from inequalities (7)-(10) being equivalent.

$$\exists \Omega \subseteq \mathcal{I} : \quad \mathrm{ect}^{\mathrm{F}}_\Omega > \mathrm{lct}_\Omega \quad \Rightarrow \text{fail} \tag{7}$$

$$\exists \Omega \subseteq \mathcal{I} : \left\lceil \frac{\max\{Cest_{\Omega'} + e_{\Omega'} \mid \Omega' \subseteq \Omega\}}{C} \right\rceil > \mathrm{lct}_\Omega \Rightarrow \text{fail} \tag{8}$$

$$\exists \Omega' \subseteq \mathcal{I} : \left\lceil \frac{Cest_{\Omega'} + e_{\Omega'}}{C} \right\rceil > \mathrm{lct}_{\Omega'} \Rightarrow \text{fail} \tag{9}$$

$$\exists \Omega' \subseteq \mathcal{I} : \quad e_{\Omega'} > C(\mathrm{lct}_{\Omega'} - \mathrm{est}_{\Omega'}) \quad \Rightarrow \text{fail} \tag{10}$$

However, since $\mathrm{ect}^{\mathrm{F}}_\Omega \leq \mathrm{ect}_\Omega$, rule (6) detects more failure cases than its fully-elastic relaxed version. This suggests to find stronger relaxations for the function ect than $\mathrm{ect}^{\mathrm{F}}$.

From [Vilím, 2009], we generalize the Edge-Finder detection rule. A precedence is detected when a set of tasks Ω, executing along a task $i \notin \Omega$, cannot meet its deadline.

$$\forall \Omega \subset \mathcal{I}, \forall i \in \mathcal{I} \setminus \Omega : \mathrm{ect}_{\Omega \cup \{i\}} > \mathrm{lct}_\Omega \Rightarrow \Omega \ll i \quad (11)$$

Since computing ect_Ω is NP-Hard, one needs to use a relaxation. The fully-elastic relaxation results in rule (3). The demonstration is similar to the one for the Overload Check.

We introduce a stronger relaxation for the function ect that we call *horizontally-elastic*. With this relaxation, a task i is allowed to consume, at any time $t \in [\mathrm{est}_i, \mathrm{lct}_i)$, between 0 and h_i units of resource. Unlike the fully-elastic relaxation, it cannot consume more than h_i units of resource. Given a set of tasks Ω, $\mathrm{ect}^{\mathrm{H}}_\Omega$ is computed using the following formulas.

Let $h_{\max}(t)$ represent the amount of resource that can be allocated to the tasks in Ω at time t. A task i can consume at most h_i units of resource at any time in its execution window $[\mathrm{est}_i, \mathrm{lct}_i)$. The resource has a capacity C.

$$h_{\max}(t) = \min\left(\textstyle\sum_{i \in \Omega | \mathrm{est}_i \leq t < \mathrm{lct}_i} h_i, C\right) \quad (12)$$

Let $h_{\mathrm{req}}(t)$ be the amount of resource required at time t by the tasks in Ω if they were all starting at their earliest starting times. In this context, a task i consumes h_i units of resource throughout the interval $[\mathrm{est}_i, \mathrm{ect}_i)$.

$$h_{\mathrm{req}}(t) = \textstyle\sum_{i \in \Omega | \mathrm{est}_i \leq t < \mathrm{ect}_i} h_i \quad (13)$$

We call overflow $ov(t)$ the energy from $h_{\mathrm{req}}(t)$ that cannot be executed at time t due to the limited capacity $h_{\max}(t)$. This overflow is accumulated over time and released when the resource is no longer saturated. Let $h_{\mathrm{cons}}(t)$ be the amount of resource that is actually consumed at time t. This amount is given by $h_{\mathrm{req}}(t)$ to which we add the previously accumulated overflow. The resource consumed is limited by $h_{\max}(t)$.

$$h_{\mathrm{cons}}(t) = \min(h_{\mathrm{req}}(t) + ov(t-1), h_{\max}(t)) \quad (14)$$

$$ov(t) = ov(t-1) + h_{\mathrm{req}}(t) - h_{\mathrm{cons}}(t) \quad (15)$$

$$ov(\min_{i \in \Omega} \mathrm{est}_i) = 0 \quad (16)$$

The earliest completion time occurs when all tasks are completed. Figure 1 shows the distribution of the energy along the time line.

$$\mathrm{ect}^{\mathrm{H}} = \max\{t \mid h_{\mathrm{cons}}(t) > 0\} + 1 \quad (17)$$

Theorem 1 shows that the horizontally-elastic relaxation is stronger than the fully-elastic one.

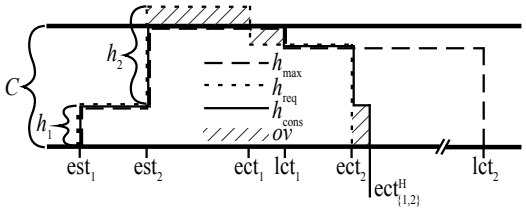

Figure 1: Fluctuation of the functions h_{req}, $h_{\max}$, h_{cons}, and ov when scheduling two tasks.

Theorem 1. *For all* $\Omega \subseteq \mathcal{I}$, $\mathrm{ect}^{\mathrm{F}}_\Omega \leq \mathrm{ect}^{\mathrm{H}}_\Omega \leq \mathrm{ect}_\Omega$

Proof. The fully-elastic relaxation requires a task i to consume e_i units of resource within the interval $[\mathrm{est}_i, \mathrm{lct}_i)$. The horizontally-elastic relaxation has the same constraint with the added restriction that no more than h_i units of resource should be consumed at any given time, which can only make a schedule finish later, hence $\mathrm{ect}^{\mathrm{F}}_\Omega \leq \mathrm{ect}^{\mathrm{H}}_\Omega$. In the CuSP, a task must consume either 0 or h_i units of resource at time $t \in [\mathrm{est}_i, \mathrm{lct}_i)$. Morever, there is no interruption. These conditions are even stronger, hence $\mathrm{ect}^{\mathrm{H}}_\Omega \leq \mathrm{ect}_\Omega$. □

Theorem 2. *The Overload Check and Edge-Finder based on the horizontally-elastic relaxation detect a superset of inconsistencies and precedences than their respective version based on the fully-elastic relaxation.*

Proof. Since $\mathrm{ect}^{\mathrm{F}}_\Omega \leq \mathrm{ect}^{\mathrm{H}}_\Omega$ for all Ω (Theorem 1), the fail condition $\mathrm{ect}^{\mathrm{F}}_\Omega > \mathrm{lct}_\Omega$ implies the fail condition $\mathrm{ect}^{\mathrm{H}}_\Omega > \mathrm{lct}_\Omega$. Similarly, $\mathrm{ect}^{\mathrm{F}}_{\Omega \cup \{i\}} > \mathrm{lct}_\Omega$ implies the detection condition $\mathrm{ect}^{\mathrm{H}}_{\Omega \cup \{i\}} > \mathrm{lct}_\Omega$. Consider the instance with $C = 2$ and four tasks whose parameters $\langle \mathrm{est}_i, \mathrm{lct}_i, p_i, h_i \rangle$ are $\langle 0, 4, 2, 1 \rangle$, $\langle 1, 4, 1, 2 \rangle$, $\langle 1, 4, 1, 2 \rangle$, and $\langle 1, 4, 1, 2 \rangle$. Only the Overload Check based on the horizontally-elastic relaxation fails. In the instance with $C = 2$ and the tasks $x : \langle 0, 5, 2, 1 \rangle$, $y : \langle 1, 5, 2, 1 \rangle$, $z : \langle 1, 5, 2, 2 \rangle$, and $w : \langle 1, 10, 2, 1 \rangle$. The precedence $\{x, y, z\} \ll w$ is only detected when using the horizontally-elastic relaxation. □

5 Resource Utilization Profile

To efficiently compute $\mathrm{ect}^{\mathrm{H}}$, we introduce a data structure called Resource Utilization Profile, or simply Profile, that stores the resource utilization over time, as in Figure 1. Similarly to [Gay *et al.*, 2015], we represent the Profile as an aggregation of juxtaposed rectangles of different lengths and heights. Rectangles are expressed with tuples $\langle \mathrm{time}, \mathrm{capacity}, \Delta_{\max}, \Delta_{\mathrm{req}} \rangle$, where *time* is the start time, *capacity* is the remaining capacity of the resource at the start time, $\Delta_{\max}$and $\Delta_{\mathrm{req}}$are two quantities initialized to zero. The ending time is the starting time of the next rectangle. These tuples are stored in a sorted linked list whose nodes are called *time points*, referring to the starting times of the rectangles. The Profile is initialized with a time point of capacity C for every distinct value of est, ect and lct. We add a sufficiently large time point to act as sentinel. Finally, while initializing the data structure, pointers are kept so that $\mathrm{T}_{\mathrm{est}_i}$, $\mathrm{T}_{\mathrm{ect}_i}$, and $\mathrm{T}_{\mathrm{lct}_i}$ return the time point associated to est_i, ect_i, and lct_i.

The algorithm `ScheduleTasks` proceeds in batch to schedule a set of tasks Θ on the profile P. The algorithm computes the functions $h_{\mathrm{req}}(t)$, $h_{\max}(t)$, $h_{\mathrm{cons}}(t)$, and $ov(t)$. To ensure a strongly polynomial running time complexity, the algorithm does not process indiviual time points, but time intervals for which the functions do not fluctuate. Line 23 sets the remaining capacity of a time point to the capacity of the resource minus the consumed capacity h_{cons}. For future uses, line 11 stores the overflow at the moment of processing the time point and line 32 resets this overflow to the minimum overflow encountered in later time points.

Algorithm 1: `ScheduleTasks`(Θ, c)

```
2  for all time point t do t.Δmax ← 0, t.Δreq ← 0
4  for i ∈ Θ do
5      Increment T_est_i.Δmax and T_est_i.Δreq by h_i
6      Decrement T_lct_i.Δmax and T_ect_i.Δreq by h_i
7  t ← P.first, ov ← 0, ect ← −∞, S ← 0, h_req ← 0
9  while t.time ≠ lct_Θ do
11     t.ov ← ov
12     l ← t.next.time − t.time
13     S ← S + t.Δmax
14     h_max ← max(S, c)
15     h_req ← h_req + t.Δreq
16     h_cons ← min(h_req + ov, h_max)
17     if 0 < ov < (h_cons − h_req) · l then
18         l ← max(1, ⌊ov / (h_cons − h_req)⌋)
20         t.insertAfter(t.time + l , t.capacity, 0, 0)
21     ov ← ov + (h_req − h_cons) · l
23     t.capacity ← c − h_cons
24     if t.capacity < c then ect ← t.next.time
25     t ← t.next
26 t.ov ← ov
27 m ← ∞
29 while t ≠ P.first and m > 0 do
30     m ← min(m, t.ov)
32     t.ov ← m
33     t ← t.previous
34 return ect, ov
```

Lemma 1. *The Profile contains at most* $4n + 1$ *time points.*

Proof. There is initially one time point for every distinct value of est, ect and lct and one sentinel. One additional time point can be created per task being scheduled on the Profile when its energy is fully spent (line 20). □

Lemma 2. `ScheduleTasks` *runs in* $O(n)$ *time.*

Proof. The loops on lines 2, 4, 9, and 29 iterate over time points. By Lemma 1 they execute $O(n)$ times. □

Theorem 3. *The Profile obtained with* `ScheduleTasks` *complies with the horizontally-elastic relaxation.*

Proof. A task i can only be executed during the interval $[\text{est}_i, \text{lct}_i)$ and can consume at most h_i units of the resource at any time point in this interval. This property is enforced with the h_{max} value. An amount of e_i energy is spent for each task i as the height of a task i is added to the h_{req} value for the interval $[\text{est}_i, \text{ect}_i)$, which has a length of p_i. □

6 Overload Check

We present an algorithm that enforces the Overload Check rule based on the horizontally-elastic relaxation, i.e. :

$$\exists \Omega \subseteq \mathcal{I} : \text{ect}^{\text{H}}_{\Omega} > \text{lct}_{\Omega} \Rightarrow \text{fail} \tag{18}$$

Algorithm 2: `OverloadCheck`$(\mathcal{I}, C)$

```
1  Θ ← ∅
2  for i ∈ I in ascending order of lct_i do
3      Θ ← Θ ∪ {i}
4      ect, ov ← ScheduleTasks(Θ, C)
5      if ect > lct_i or ov > 0 then fail
```

The algorithm `OverloadCheck` is essentially the same as Vilím's [2009] except for the value of ect that is computed using `ScheduleTasks`. The algorithm also fails if some overflow was unspent beyond time lct_{Θ}.

Lemma 3. `OverloadCheck` *runs in* $O(n^2)$ *time.*

Proof. The linear time algorithm `ScheduleTasks` is called n times. □

7 Edge-Finder Detection

We introduce an algorithm that enforces the Edge-Finder rule (11) based on the horizontally-elastic relaxation.

Like Vilím's [2009] algorithm, `Detection` iterates over all tasks in non-increasing order of lct. On each iteration, the function `ScheduleTasks` schedules on an empty Profile the left cut Θ of the current task. `DetectPrecedences` tests for precedence detection the tasks in Λ. The function `DetectPrecedences` returns all tasks $j \in \Lambda$ for which the rule (11) detects $\Theta \ll j$, but requires all tasks in Λ to have the same height. This is why the loop on line 8 of `Detection` iterates over all heights in Λ. When a precedence is detected with task j, it dominates all precedences that could be further discovered, so j is removed from Λ.

The algorithm `DetectPrecedences` iterates over the profile in backward order and keeps track of the remaining resource capacity over the time intervals. While processing time t, the algorithm certifies that a task of height h and energy at most e starting at time t can finish at or before lct. Moreover, it keeps track of the overflow ov that a task starting before time t can spend after time t. When the algorithm processes a time t for which there is a task $j \in \Lambda$ such that $\text{est}_j = t$ and $e_j - \max(0, h_j(\text{ect}_j - \text{lct})) > e$, it infers that task j cannot finish before time lct, i.e. $\Theta \ll j$. The term

Algorithm 3: `Detection`$(\mathcal{I}, C)$

```
1  Prec ← ∅, Θ ← I, Λ ← ∅
2  for t ∈ {lct_i | i ∈ I} \ {min_{i∈I} lct_i} in desc. ord. do
3      Θ ← Θ \ {j | lct_j = t}
4      Λ ← Λ ∪ {j | lct_j = t}
5      ect, ov ← ScheduleTasks(Θ, C)
6      if ov > 0 then fail
8      for h ∈ {h_i | i ∈ Λ} do
9          Λ^h ← {i ∈ Λ | h_i = h}
10         Ω ← DetectPrecedences(Θ, Λ^h, h, lct_Θ)
11         Prec ← Prec ∪ {Θ ≪ j | j ∈ Ω}
12         Λ ← Λ \ Ω
```

Algorithm 4: `DetectPrecedences`$(\Theta, \Lambda^h, h, \mathrm{lct})$

```
1  for all time point t do t.Δmax ← 0
2  for i ∈ Θ do
3      Decrement T_{est_i}.Δmax by h_i
4      Increment T_{lct_i}.Δmax by h_i
5  minest ← min_{i∈Λ^h} est_i
6  t ← getNode(lct).previous
7  Ω ← ∅, e ← 0 , ov ← 0, h_max ← h
8  while t.time ≥ minest do
9      l ← t.next.time − t.time
10     h_max ← h_max + t.next.Δmax
11     c ← min(t.capacity, h_max − (C − t.capacity))
12     e ← e + l · min(c, h) + max(0, min(ov, (h − c)l))
13     ov ← max(0, ov + l(c − h))
15     Ω ← Ω ∪ {j ∈ Λ^h |
16           est_j = t.time, e_j − min(0, h_j(ect_j − lct)) > e}
17     t ← t.previous
18 return Ω
```

Algorithm 5: `Adjustment`(Prec, C)

```
1  for Θ ≪ i ∈ Prec do
2      ect, ov ← ScheduleTasks(Θ, C − h_i)
3      est_i ← max(est_i, ComputeBound(i, Θ, ov))
```

$\max(0, h_j(\mathrm{ect}_j - \mathrm{lct}))$ represents the energy of task j that cannot be spent within $[\mathrm{est}_\Theta, \mathrm{lct}_\Theta)$.

Lemma 4. *`DetectPrecedences` runs in $O(n)$ time.*

Proof. All loops iterate $O(n)$ times, once per time point (Lemma 1). Tasks in Λ are added at most once to Ω on line 15. Therefore `DetectPrecedences` runs in $O(n)$. □

Lemma 5. *`Detection` runs in $O(kn^2)$ time, where k is the number of distinct heights.*

Proof. `Detection` calls `ScheduleTasks` for each task. For all distinct heights, it calls `DetectPrecedences`. Hence a complexity of $O(n(n + kn)) = O(kn^2)$. □

8 Edge-Finder Adjustment

We introduce a stronger adjustment algorithm that utilizes the strength of the horizontally-elastic relaxation. Given a precedence relation $\Theta \ll i$ discovered during the detection phase, `Adjustment` computes the new value for est_i.

Let the bottom (resp. upper) part of the resource be a portion with capacity $C - h_i$ (resp. h_i). `Adjustment` iterates through all detected precedences $\Theta \ll i$ and schedules the tasks in Θ on the bottom part of the resource. Because of this restriction on the capacity, the energy of the tasks is not fully scheduled which results in an overflow ov returned by `ScheduleTasks`. This overflow is an accumulation of small overflows that `ScheduleTasks` encoded on the profile as follows. For each time point t, $t.ov$ indicates how much overflow, that contributed to the final overflow, was accumulated on the time interval $(-\infty, t.time]$. All this overflow must be scheduled on the upper part of the resource. `ComputeBound` simulates the execution of `ScheduleTasks` that schedules the tasks in Θ. However, it only allows $t.ov$ units of energy on the time interval $(-\infty, t.time]$ to be scheduled on the upper part of the resource. Term d_{total} represents the energy that was scheduled on the upper part in the previous iterations, while d represents the energy that is about to be scheduled on the upper part in the current iteration. When $ov_{\max}$ units of energy are scheduled on the upper part, the algorithm stops and returns the time when this event occurs. This is where est_i is adjusted.

Lemma 6. *`Adjustment` runs in $O(n^2)$.*

Proof. `ComputeBound` iterates $O(n)$ times (Lemma 1) and each iteration executes in $O(1)$. `ScheduleTasks` also runs in linear time. `Adjustment` calls these algorithms $O(n)$ times for a total complexity of $O(n^2)$. □

Consider a detected precedence $\Theta \ll i$. Let $\mathrm{adj}_i^{\mathrm{F}}$ ($\mathrm{adj}_i^{\mathrm{H}}$) be the earliest starting time of task i after being adjusted using the classic rule (4) (the algorithm `Adjustment`).

Theorem 4. *For all precedences $\Theta \ll i$, $\mathrm{adj}_i^{\mathrm{F}} \leq \mathrm{adj}_i^{\mathrm{H}}$*

Proof. Both adjustments assign as much energy as possible on the bottom part of the resource and the remaining energy onto the upper part. But the horizontally-elastic relaxation

Algorithm 6: `ComputeBound`$(i, \Theta, ov_{\max})$

```
1  for all time point t do t.Δmax ← 0, t.Δreq ← 0
2  for i ∈ Θ do
3      Increment T_{est_i}.Δmax and T_{est_i}.Δreq by h_i
4      Decrement T_{lct_i}.Δmax and T_{ect_i}.Δreq by h_i
5  t ← min{t | t.ov > 0}, ov ← 0 , d_total ← 0
6  S ← 0, h_req ← 0
7  while t.next ≠ null do
8      l ← t.next.time − t.time
9      S ← S + t.Δmax
10     h_max ← min(S, C)
11     h_req ← h_req + t.Δreq
12     h_cons ← min (h_req + ov, h_max)
13     if 0 < ov < (h_cons − h_req) · l then
14         l ← max ( ⌊ ov / (h_cons − h_req) ⌋ , 1)
15     d ← (h_cons − C + h_i) · l
16     d ← max(0, min(d, ov_max − d_total, t.next.ov − d_total)
17     if d_total + d = ov_max then
18         return min (t.next.time, t.time + ⌈ d / (h_cons − (C − h_i)) ⌉ )
19     d_total ← d_total + d
20     ov ← ov + (h_req − h_cons) · l
21     if t.time + l < t.next.time then
22         t.time ← t.time + l
23     else t ← t.next
24 return −∞
```

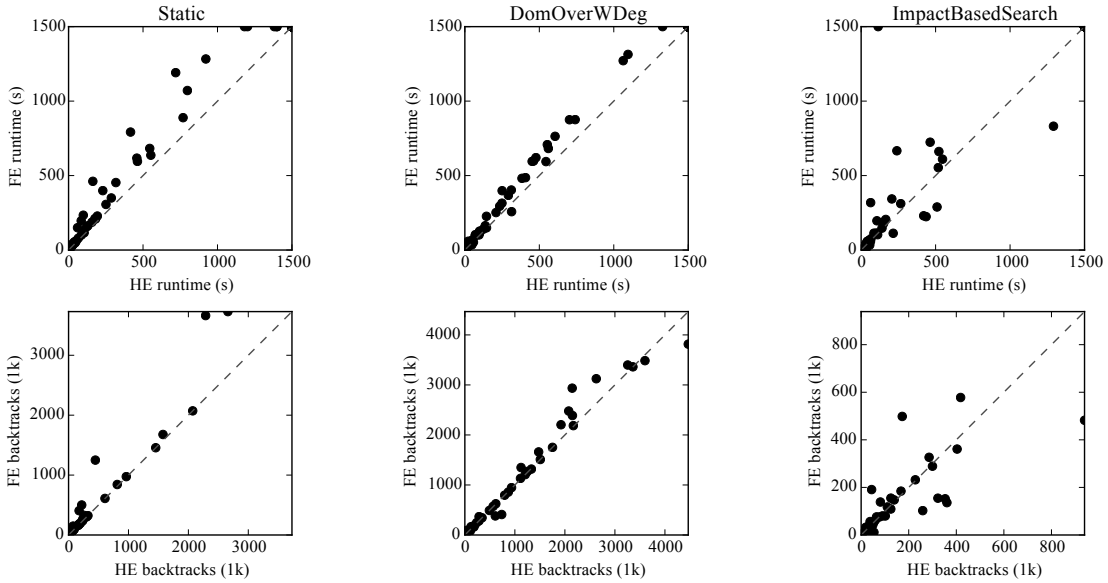

Figure 2: Runtimes and backtracks comparison

limits to h_i the amount of energy spent by a task i at any given time which shifts the energy later on the schedule. The fully-elastic relaxation consumes the bottom part of the resource entirely and packs the remaining energy as soon as possible on the upper part. The horizontally-elastic relaxation might not fully consume the bottom part and does not necessarily pack the remaining energy at the earliest time on the upper part. Consider the instance with $C = 3$ and five tasks whose parameters $\langle \mathrm{est}_i, \mathrm{lct}_i, p_i, h_i \rangle$ are $x : \langle 0, 4, 2, 1 \rangle$, $y : \langle 1, 4, 1, 3 \rangle$, $z : \langle 2, 4, 1, 3 \rangle$, $w : \langle 2, 4, 1, 1 \rangle$, and $v : \langle 1, 10, 3, 1 \rangle$. We get $\mathrm{adj}_v^{\mathrm{F}} = 2 < \mathrm{adj}_v^{\mathrm{H}} = 3$ for the precedence $\{x, y, z, w\} \ll v$. □

9 Experimental results

We implemented the algorithms for the Choco 2 solver and tried them on the benchmarks BL [Baptiste and Le Pape, 2000] and PSPLib [Kolisch and Sprecher, 1997] of Resource Constrained Project Scheduling Problems (RCPSP). This problem has multiple resources of varied capacities on which the tasks, subject to precedence constraints, are simultaneously executed. We minimize the makespan. The base model has one starting time variable per task, one makespan variable, and one CUMULATIVE constraint per resource for which the Time-Table rule is enforced [Beldiceanu and Carlsson, 2002]. From this common core, we create two models that are compared against each other. In the *horizontally-elastic model*, we enforce the Overload Check and Edge-Finder rules as explained in the previous sections. In the *fully-elastic model*, we rather post our implementation of a constraint that enforces the Overload Check and Edge-Finder rules as described by [Vilím, 2009]. We used three different branching heuristics: a static variable and value ordering, DomOverWDeg [Boussemart *et al.*, 2004], and Impact Based Search [Refalo, 2004]. All experiments were run on an Intel Xeon X5560 2.667GHz quad-core processor.

Figure 2 compares runtimes and backtracks for both models. We only report instances that were solved within 25 minutes. For the highly-cumulative instances from the BL benchmark, our method gives a speedup for 85%, 82% and 63% of the instances using the static, the DomOverWDeg and the Impact Based Search heuristics. The Impact Based Search leads to the lesser improvement since it tends not to branch on values that would be filtered. For the highly-disjunctive instances from the PSPLib benchmark, we notice a lesser improvement of runtimes since the instances are generally easier to resolve. Our method still allows a speedup for 75% of the instances for all three heuristics.

The horizontally-elastic relaxation grants a significant improvement in the runtimes, although not as significant for the backtracks. This is explained by Vilím's algorithm that processes a large number of detected precedences that do not lead to adjustments. We observed that our algorithm only discovers precedences that lead to adjusments. Moreover, our algorithm processes each precedence in linear time.

10 Conclusion

We generalized the Overload Check and Edge-Finder rules using a function of earliest completion time (ect). We introduced a stronger relaxation of ect. We presented an innovative data structure used in two new algorithms that enforce the Overload Check and Edge-Finder rules. Experimental results demonstrate the effectiveness of our method. In fact, our algorithms solved many RCPSP instances with fewer backtracks and in faster times than the state-of-the-art algorithms.

References

[Baptiste and Le Pape, 2000] Philippe Baptiste and Claude Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, 5(1-2):119–139, 2000.

[Baptiste *et al.*, 2001] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*, volume 39. Springer Science & Business Media, 2001.

[Beldiceanu and Carlsson, 2002] Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In *Principles and Practice of Constraint Programming-CP 2002*, pages 63–79, 2002.

[Boussemart *et al.*, 2004] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.

[Buyya *et al.*, 2005] Rajkumar Buyya, Manzur Murshed, David Abramson, and Srikumar Venugopal. Scheduling parameter sweep applications on global grids: a deadline and budget constrained cost–time optimization algorithm. *Software: Practice and Experience*, 35(5):491–512, 2005.

[Fahimi and Quimper, 2014] Hamed Fahimi and Claude-Guy Quimper. Linear-time filtering algorithms for the disjunctive constraint. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2637–2643, 2014.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and intractability*. W.H. Freeman, 1979.

[Gay *et al.*, 2015] Steven Gay, Renaud Hartert, and Pierre Schaus. Simple and scalable time-table filtering for the cumulative constraint. In *Principles and Practice of Constraint Programming*, pages 149–157, 2015.

[Harjunkoski *et al.*, 2014] Iiro Harjunkoski, Christos T Maravelias, Peter Bongers, Pedro M Castro, Sebastian Engell, Ignacio E Grossmann, John Hooker, Carlos Méndez, Guido Sand, and John Wassick. Scope for industrial applications of production scheduling models and solution methods. *Computers & Chemical Engineering*, 62:161–193, 2014.

[Kameugne *et al.*, 2014] Roger Kameugne, Laure Pauline Fotso, Joseph Scott, and Youcheu Ngo-Kateu. A quadratic edge-finding filtering algorithm for cumulative resource constraints. *Constraints*, 19(3):243–269, 2014.

[Kolisch and Sprecher, 1997] Rainer Kolisch and Arno Sprecher. Psplib-a project scheduling problem library: Or software-orsep operations research software exchange program. *European journal of operational research*, 96(1):205–216, 1997.

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

[Mercier and Van Hentenryck, 2008] Luc Mercier and Pascal Van Hentenryck. Edge finding for cumulative scheduling. *INFORMS Journal on Computing*, 20(1):143–153, 2008.

[Refalo, 2004] Philippe Refalo. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming–CP 2004*, pages 557–571. Springer, 2004.

[Vilím, 2009] Petr Vilím. Edge finding filtering algorithm for discrete cumulative resources in $O(kn \log n)$. In *Principles and Practice of Constraint Programming-CP 2009*, pages 802–816. Springer, 2009.

[Vilím, 2011] Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 230–245. Springer, 2011.

[Wolf and Schrader, 2006] Armin Wolf and Gunnar Schrader. $O(n \log n)$ overload checking for the cumulative constraint and its application. In *Declarative Programming for Knowledge Management*, pages 88–101. Springer, 2006.