Noname manuscript No.
(will be inserted by the editor)

Linear-Time Filtering Algorithms for the Disjunctive Constraint and a Quadratic Filtering Algorithm for the Cumulative Not-First Not-Last

Hamed Fahimi · Yanick Ouellet · Claude-Guy Quimper

Received: date / Accepted: date

Abstract We present new filtering algorithms for DISJUNCTIVE and CUMULATIVE constraints, each of which improves the complexity of the state-of-the-art algorithms by a factor of $\log n$. We show how to perform Time-Tabling and Detectable Precedences in linear time on the DISJUNCTIVE constraint. Furthermore, we present a linear-time Overload Checking for the DISJUNCTIVE and CUMULATIVE constraints. Finally, we show how the rule of Not-first/Not-last can be enforced in quadratic time for the CUMULATIVE constraint. These algorithms rely on the union find data structure, from which we take advantage to introduce a new data structure that we call it time line. This data structure provides constant time operations that were previously implemented in logarithmic time by the Θ-tree data structure. Experiments show that these new algorithms are competitive even for a small number of tasks and outperform existing algorithms as the number of tasks increases. We also show that the time line can be used to solve specific scheduling problems.

1 Introduction

Constraint programming offers numerous ways to model and solve scheduling problems. The DISJUNCTIVE constraint allows to model problems, where the tasks cannot be executed concurrently. The CUMULATIVE constraint models the problems where a limited number of tasks can execute simultaneously. These constraints are used in many applications. For instance, they are used to solve continuous casting

A preliminary version of this paper appears as [8]

Claude-Guy Quimper
Tel.: +1-418-656-2131 ext. 2099
E-mail: Claude-Guy.Quimper@ift.ulaval.ca

Yanick Ouellet
Tel: +1-418-656-2131 ext. 4799
E-mail: yanick.ouellet.2@ulaval.ca

Hamed Fahimi
Tel: +1-418-656-2131 ext. 4799
E-mail: hamed.fahimi.1@ulaval.ca

scheduling problems [11], cargo assembly planning problems [5], university time tabling [12], and even carpet cutting problems [25].

Multiple filtering algorithms to prune the search space come with these constraints. Due to the vast number of calls to these algorithms throughout the search, it is essential to design them as efficient as possible. The contribution of data structures to the efficiency of these algorithms is indispensable.

For the All-Different constraint, that is a special case of Disjunctive and Cumulative constraints, Puget [21] proposes an $O(n \log n)$ filtering algorithm. The factor $\log n$ stems from the operations achieved on a balanced tree of depth $\log n$. López-Ortiz et al. [16] and Melhorn and Thiel [18] present linear time algorithms that both use union find data structures to achieve equivalent operations. Vilím [30] proposes new data structures, called Θ-tree and Θ-λ tree, that are balanced trees of depth $\log n$. These data structures led to filtering algorithms for the Disjunctive constraint, including filtering algorithms based on Overload Checking, Not-first/Not-last, and Detectable Precedences [34]. Furthermore, filtering algorithms for the Cumulative constraint were achieved for Edge-Finding [31], Extended-Edge-Finding, Time-Tabling-Extended-Edge-Finding [20], and Not-first/Not-last [14]. These algorithms have a $\log n$ factor in their running time complexities that originates from the balanced tree. In order to acquire a faster running time complexity, we propose to modify some of these algorithms by using a union find data structure, as it was done for All-Different.

The paper is organized as follows. In Section 2, we review the Disjunctive and Cumulative constraints, as well as three filtering techniques: Time-Tabling, Not-first/Not-last, and Detectable Precedences. In addition, we go over the Overload Checking, which works as a consistency test to provide a necessary condition for the feasibility of a set of tasks. Section 3 presents algorithmic preliminaries. In Section 4 we propose the time line data structure. In Section 5, we introduce new algorithms: a linear time algorithm for Time-Tabling (Section 5.1), a linear time Overload Checking (Section 5.2), a Not-first/Not-last (Section 5.3), and an algorithm applying the Detectable Precedences rules (Section 5.4). We also present algorithms that solve simple scheduling problems in linear time (Section 5.5). Thereafter, Section 6 presents the experimental results which verify the efficiency of our algorithms. Section 7 concludes with a summary of our contributions.

This paper is an extension of a previous paper [8]. We present here, for the first time, an algorithm that enforces the Not-first/Not-last filtering rule in quadratic time, a linear time algorithm that minimizes total delay, and a linear time algorithm that minimizes maximum lateness.

2 The Disjunctive and Cumulative constraints

We consider the scheduling problem where n tasks (denoted $\mathcal{I} = \{1, \dots, n\}$) compete to be executed without interruption, on a resource of capacity $\mathcal{C}$. Every task 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 height $h_i \in \mathbb{Z}^+$. The height of a task is the rate at which it consumes the resource and thus $h_i \leq \mathcal{C}$. From these properties, one can compute the latest starting time $\mathrm{lst}_i = \mathrm{lct}_i - p_i$ and the earliest completion time $\mathrm{ect}_i = \mathrm{est}_i + p_i$. The *energy* of a task, computed with $\mathrm{e}_i = h_i p_i$, is the amount of

resource it consumes throughout its execution. The notions of the earliest starting time, the latest completion time, the processing time, and the energy can be generalized to a set of tasks $\emptyset \subset \Omega \subseteq \mathcal{I}$.

$$\mathrm{est}_\Omega = \min_{i \in \Omega} \mathrm{est}_i \qquad \mathrm{lct}_\Omega = \max_{i \in \Omega} \mathrm{lct}_i \qquad p_\Omega = \sum_{i \in \Omega} p_i \qquad \mathrm{e}_\Omega = \sum_{i \in \Omega} \mathrm{e}_i$$

For empty sets, we have $\mathrm{est}_\emptyset = \infty$ and $\mathrm{lct}_\emptyset = -\infty$.

A resource is said to be *cumulative* if it can execute multiple tasks simultaneously. The sum of the heights of the tasks executing at a given time cannot exceed $\mathcal{C}$. A resource is *disjunctive* when it can only execute one task at a time. In such a case, we have $h_i + h_j > \mathcal{C}$ for any pair of distinct tasks $i, j \in \mathcal{I}$.

In the disjunctive case, a lower bound on the completion time of a set of tasks $\Omega \subseteq \mathcal{I}$ can be obtained by scheduling the tasks with preemption. We denote ect_Ω to stand for the earliest completion time of the tasks in Ω when scheduled with preemption. Similarly, lst_Ω signifies the latest starting time of the tasks in Ω when scheduled with preemption.

$$\mathrm{ect}_\Omega = \max_{\emptyset \subset \Theta \subseteq \Omega} (\mathrm{est}_\Theta + p_\Theta) \qquad \mathrm{lst}_\Omega = \min_{\emptyset \subset \Theta \subseteq \Omega} (\mathrm{lct}_\Theta - p_\Theta) \tag{1}$$

For empty sets, we have $\mathrm{ect}_\emptyset = -\infty$ and $\mathrm{lst}_\emptyset = \infty$.

For the cumulative case, Vilím [32] defines the *energy envelope* of a set of tasks $\Omega \subseteq \mathcal{I}$ as follows

$$\mathrm{Env}_\Omega = \max_{\Theta \subseteq \Omega} (\mathcal{C}\, \mathrm{est}_\Theta + \mathrm{e}_\Theta). \tag{2}$$

The envelope is used to compute the earliest completion time of a set of tasks, when scheduled with preemption.

$$\mathrm{ect}_\Omega = \left\lceil \frac{\mathrm{Env}_\Omega}{\mathcal{C}} \right\rceil \tag{3}$$

To simplify the algorithms, we sometimes reduce a problem with a cumulative resource with capacity $\mathcal{C} > 1$ to a problem with a resource of unit capacity $\mathcal{C} = 1$. A task i is transformed into a task i' with $\mathrm{est}_{i'} = \mathcal{C} \cdot \mathrm{est}_i$, $\mathrm{lct}_{i'} = \mathcal{C} \cdot \mathrm{lct}_i$, $p_{i'} = \mathrm{e}_i$, and $h_{i'} = 1$. Let Ω be a set of tasks in the original problem and Ω' be the transformed tasks corresponding to those in Ω. The following lemma shows how to compute the energy enveloppe of the set Ω.

Lemma 1 *The energy enveloppe of a set of tasks Ω in the original problem is equal to the earliest completition time of the transformed tasks when scheduled with preemption, i.e.* $\mathrm{Env}_\Omega = \mathrm{ect}_{\Omega'}$.

Proof The proof follows the properties of fully elastic scheduling problems presented in [2]. It is summarized with the equalities $\mathrm{ect}_{\Omega'} = \max_{\emptyset \subset \Theta' \subseteq \Omega'} (\mathrm{est}_{\Theta'} + p_{\Theta'}) = \max_{\emptyset \subset \Theta \subseteq \Omega} (\mathcal{C} \cdot \mathrm{est}_\Theta + \mathrm{e}_\Theta) = \mathrm{Env}_\Omega$. □

Let the variable S_i denote the starting time of task i with the domain $\mathrm{dom}(S_i) = [\mathrm{est}_i, \mathrm{lst}_i]$. The constraint $\textsc{Disjunctive}([S_1, \ldots, S_n], \mathbf{p})$ is satisfied when $S_i + p_i \leq S_j \vee S_j + p_j \leq S_i$ for all pairs of tasks $i \neq j$. A solution to the $\textsc{Disjunctive}$

constraint is a solution to the disjunctive scheduling problem. The constraint CUMULATIVE$([S_1, \dots, S_n], \mathbf{p}, \mathbf{h}, \mathcal{C})$ holds if and only if

$$\forall t : \sum_{i:S_i \leq t < S_i + p_i} h_i \leq \mathcal{C}$$

The CUMULATIVE constraint implies that the total resource consumption rate underway at time t does not exceed $\mathcal{C}$.

It is NP-complete to determine whether the DISJUNCTIVE and CUMULATIVE constraints are satisfiable and therefore it is NP-hard to enforce bound consistency on them [1]. Notwithstanding, there are polynomial time algorithms for specializations to these constraints. For instance, when $\mathcal{C} = h_i = p_i = 1$, the CUMULATIVE constraint is interpreted as an ALL-DIFFERENT constraint and bound consistency can be achieved in linear time [16,18]. When $p_i = p_j$ for all $i, j \in \mathcal{I}$, the DISJUNCTIVE constraint becomes an INTER-DISTANCE constraint and bound consistency can be achieved in quadratic time [22].

Even though it is NP-Hard to filter the DISJUNCTIVE and CUMULATIVE constraints, there exist pruning rules that can be enforced in polynomial time. All the rules we present aim at delaying the earliest starting time of the tasks. To advance the latest completion time, one can create the symmetric problem where task i is transformed into a task i' such that $\mathrm{est}_{i'} = -\mathrm{lct}_i$, $\mathrm{lct}_{i'} = -\mathrm{est}_i$, $h_{i'} = h_i$, and $p_{i'} = p_i$. Delaying the earliest starting time in the symmetric problem prunes the latest completion time in the original problem.

Following Vilim's notations [30,31], we establish the following convention. Let $i \in \mathcal{I}$ be a task and $\Theta \subseteq \mathcal{I} \setminus \{i\}$ be a set of tasks. The relation $\Theta \lessdot i$ denotes that the task i must *complete* once all tasks in Θ have completed, whereas the relation $\Theta \ll i$ denotes that the task i must *start* once all tasks in Θ have completed. The first precedence relation is detected when filtering the CUMULATIVE constraint, while the second precedence relation is detected when filtering the DISJUNCTIVE constraint.

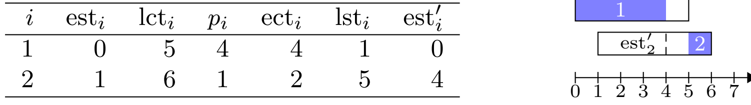
2.1 Time-Tabling for the DISJUNCTIVE constraint

If a task $i \in \mathcal{I}$ with $\mathrm{lst}_i < \mathrm{ect}_i$ exists, the time window $[\mathrm{lst}_i, \mathrm{ect}_i)$ is said to be the *compulsory part* of i, as such a task must execute within this interval. This is the key observation behind the *Time-Tabling* rule. If there exists a task i with a compulsory part and there exists a task j that satisfies $\mathrm{ect}_j > \mathrm{lst}_i$, inevitably j must execute after i, i.e. $i \ll j$.

$$\mathrm{lst}_i < \mathrm{ect}_i \wedge \mathrm{lst}_i < \mathrm{ect}_j \Rightarrow \mathrm{est}'_j = \max(\mathrm{est}_j, \mathrm{ect}_i) \quad (4)$$

Example 1 Figure 1 corresponds to a set of tasks $\mathcal{I} = \{1, 2\}$ that must execute on a disjunctive resource of capacity $\mathcal{C} = 1$. Task 1 has the compulsory part $[1, 4)$. Since $\mathrm{lst}_1 = 1 < \mathrm{ect}_2 = 2$, the earliest starting time of the task 2 is adjusted to $\mathrm{est}_1 = \max(1, 4) = 4$ by (4).

Several algorithms apply the Time-Tabling rules [3,4,6,10,15,17,20]. Most of them were designed for the CUMULATIVE constraint but can also be used for the more restrictive case of the DISJUNCTIVE constraint.



i	est_i	lct_i	p_i	ect_i	lst_i	est'_i
1	0	5	4	4	1	0
2	1	6	1	2	5	4

Fig. 1: A set of tasks $\mathcal{I} = \{1, 2\}$ to execute on a disjunctive resource of capacity $\mathcal{C} = 1$. Due to the compulsory part $[1, 4)$ for task 1, Time-Tabling detects $1 \ll 2$ and adjusts $\text{est}'_2 = 4$.

2.2 Overload Checking

The *Overload Checking* does not filter the search space and rather detects inconsistencies and triggers backtracks during the search. For the DISJUNCTIVE constraint, the Overload Checking fails when a set of tasks $\Omega \subseteq \mathcal{I}$ whose total processing time exceeds the time allowed for the tasks to execute.

$$p_\Omega > \text{lct}_\Omega - \text{est}_\Omega \Rightarrow \text{Fail} \tag{5}$$

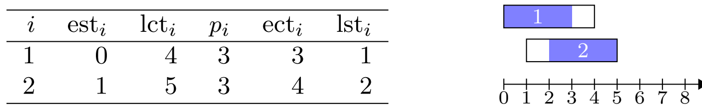
This can be rewritten in a simpler way.

$$\text{ect}_\Omega > \text{lct}_\Omega \Rightarrow \text{Fail} \tag{6}$$

The Overload Checking is generalized for the CUMULATIVE constraint as follows.

$$e_\Omega > \mathcal{C}(\text{lct}_\Omega - \text{est}_\Omega) \Rightarrow \text{Fail} \tag{7}$$

Example 2 Consider the tasks from Figure 2 that must execute on a disjunctive resource of capacity $\mathcal{C} = 1$. The Overload Checking returns a failure with the set $\Omega = \{1, 2\}$, as the total processing time $p_\Omega = 6$ is more than 5, i.e. the amount of time available in the interval $[0, 5)$.



i	est_i	lct_i	p_i	ect_i	lst_i
1	0	4	3	3	1
2	1	5	3	4	2

Fig. 2: A set of tasks $\mathcal{I} = \{1, 2\}$ to execute on a disjunctive resource of capacity $\mathcal{C} = 1$. The Overload Checking detects a failure with the set $\Omega = \{1, 2\}$.

Vilím [29] and Wolf et al. [35] propose algorithms that run in $O(n \log n)$ for this consistency test.

2.3 Not-first/Not-last

The filtering rule *not-first* determines whether a task $i \in \mathcal{I}$ must execute after at least one task in a set $\Omega \subseteq \mathcal{I} \setminus \{i\}$. The filtering rule for the DISJUNCTIVE constraint is as follows

$$\text{lct}_\Omega - \text{est}_i < p_\Omega + p_i \Rightarrow \neg(i \lessdot \Omega) \tag{8}$$

and it is generalized for the CUMULATIVE constraint as follows

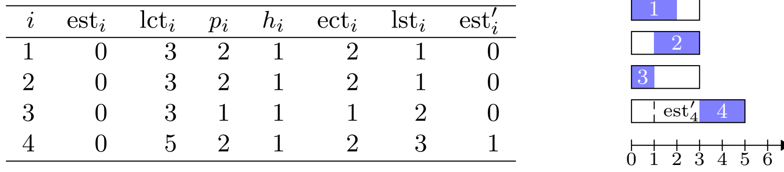
$$\text{e}_\Omega + h_i(\min(\text{ect}_i, \text{lct}_\Omega) - \text{est}_\Omega) > \mathcal{C}(\text{lct}_\Omega - \text{est}_\Omega) \Rightarrow \neg(i \lessdot \Omega). \tag{9}$$

Upon the detection of the not-first relation $\neg(i \lessdot \Omega)$, the rule applies the adjustment

$$\text{est}'_i = \max(\text{est}_i, \min_{j \in \Omega} \text{ect}_j) \tag{10}$$

The *not-last* rule is symmetric to the not-fist rule and prunes the latest completion times of tasks.

Example 3 For the tasks of Figure 3, the Not-first detects that the total energy of $e_\Omega + h_4(\min(\text{ect}_4, \text{lct}_\Omega) - \text{est}_\Omega) = 5 + 1 \cdot (2 - 0) = 7$ is greater than the energy of 6 available between lst_Ω and lct_Ω for $\Omega = \{1, 2, 3\}$. Thus, task 4 can not precede Ω and est_4 is filtered to $\min_{j \in \Omega} \text{ect}_j = 1$.



i	est_i	lct_i	p_i	h_i	ect_i	lst_i	est'_i
1	0	3	2	1	2	1	0
2	0	3	2	1	2	1	0
3	0	3	1	1	1	2	0
4	0	5	2	1	2	3	1

Fig. 3: A set of tasks $\mathcal{I} = \{1, 2, 3, 4\}$ to execute on a cumulative resource of capacity $\mathcal{C} = 2$. The Not-first detects $\neg(4 \ll \{1, 2, 3\})$ and adjusts $\text{est}'_4 = 1$

Nuijten [19] presents the Not-first/Not-last rule for the CUMULATIVE constraint. Schutt et al. [26] correct Nuijten's algorithm and provide a new algorithm running in $O(n^3 \log(n))$. Further, Kameugne et al. [14] present a sound algorithm which runs in $O(n^2 \log(n))$. Finally, Vilím [30] presents an algorithm with time complexity $O(n \log n)$ for the DISJUNCTIVE constraint.

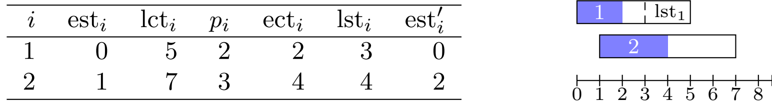
2.4 Detectable Precedences

Detectable Precedences is a technique suited for the DISJUNCTIVE constraint. For two distinct tasks $i, j \in \mathcal{I}$, when $\text{ect}_i > \text{lst}_j$ holds, we say that the precedence $j \ll i$ is *detectable*. Notice that the discrepancy between Detectable Precedences and Time-Tabling, introduced in Section 2.1, is that the task j does not necessarily

have a compulsory part. The Detectable Precedences technique consists of finding, for a task i, the set of tasks $\Omega_i = \{j \in \mathcal{I} \setminus \{i\} \mid \text{ect}_i > \text{lst}_j\}$ for which there exists a detectable precedence with i. Once such a set is discovered, one can delay the earliest starting time of i up to ect_{Ω_i}.

$$\text{est}'_i = \max(\text{est}_i, \text{ect}_{\Omega_i}) \tag{11}$$

Example 4 For the tasks of Figure 4, $\Omega_2 = \{1\}$ and $\text{ect}_{\Omega_2} = 2$. According to (11), the adjustment rule of Detectable Precedences yields $\text{est}'_2 = 2$.



i	est_i	lct_i	p_i	ect_i	lst_i	est'_i
1	0	5	2	2	3	0
2	1	7	3	4	4	2

Fig. 4: A set of tasks $\mathcal{I} = \{1, 2\}$ to execute on a disjunctive resource of capacity $\mathcal{C} = 1$. Dectectable Precedences detects $1 \ll 2$ and adjusts $\text{est}'_2 = 2$

Vilím [28] proposes this filtering technique and he later improves it in [29] to obtain an algorithm with a running time complexity of $O(n \log n)$.

3 Algorithmic Preliminaries

3.1 Sorting

Let $\mathcal{I}_{\text{est}}$, $\mathcal{I}_{\text{lst}}$, $\mathcal{I}_{\text{ect}}$, $\mathcal{I}_{\text{lct}}$, $\mathcal{I}_p$and $\mathcal{I}_d$ respectively denote the ordered sets of tasks $\mathcal{I}$ sorted by earliest starting times, latest starting times, earliest completion times, latest completion times, processing times, and due dates. We show how these sets can be sorted in linear time $O(n)$. Let w be the word-size of the processor and all time points are encoded with w-bit integers. This assumption is supported by the fact that a word of $w = 32$ bits is sufficient to encode all time points, with a precision of a second, within a period longer than a century. This is sufficient for most industrial applications. An algorithm such as *radix sort* can sort the time points in time $O(wn)$ which is linear when w is constant. Moreover, since the filtering algorithms are called multiple times with very similar instances, the *insertion sort* can resort the sets in $O(kn)$ time, where k is the number of tasks whose parameters changed since the last execution of the filtering algorithm (see [7]). Since k is usually constant, it turns out that insertion sort has a linear time behavior.

Using the insertion sort provides a way to make the filtering algorithms incremental, i.e. to take advantage of previous computations. In fact, using the insertion sort is the only strategy we use to make the algorithms incremental. All other data structures are recomputed from scratch.

3.2 Union-Find Data Structure

The new algorithms we present rely on the Union-Find data structure. The function `UnionFind`(n) initializes n disjoint sets $\{0\}, \{1\}, \ldots, \{n-1\}$ in $O(n)$ steps. The

function `Union`(a, b) merges the set that contains the element a with the set that contains the element b. The functions `FindSmallest`(a) and `FindGreatest`(a) return the smallest and greatest element of the set that contains a. These three functions run in $O(\alpha(n))$ steps, where α is Ackermann's inverse function. It is explained in [7] how to implement this data structure using trees. The smallest and greatest element of each set can be stored in the root of these trees. This implementation is the fastest in practice. However, we use this data structure in a very specific context where the function `Union`(a, b) is called only when `FindGreatest`$(a)+1 =$ `FindSmallest`(b). Such a restriction allows to use the Union-Find data structure as presented by Gabow et al. [9] who implement the functions `Union`(a, b), `FindSmallest`(a) and `FindGreatest`(a) in constant time. This implementation is the fastest in theory, but not in practice due to a large hidden constant.

4 The Time Line Data Structure

We introduce the *time line* data structure. This data structure is initialized with an empty set of tasks $\Theta = \emptyset$. It is possible to add, in constant time, a task to Θ and to compute, in constant time, the earliest completion time ect_Θ. Vilím [29] proposes the Θ-tree data structure that supports the same operations. It differs in two points from the time line. First, inserting a task in a Θ-tree requires $O(\log n)$ steps. Second, removing a task from a Θ-tree is done in $O(\log n)$ steps, whereas this operation is not supported in a time line. The time line is therefore faster than a Θ-tree, but can only be used in the contexts where the removal of a task is not required. Table 1 shows the advantage of time line as well as its limitation compared with the Θ-tree .

Operation	**$\Theta-$tree**	**Time line**
Initialization	$O(n)$	$O(n)$
Adding a task to the schedule	$O(\log(n))$	O(1)
Computing the earliest completion time	$O(1)$	$O(1)$
Removing a task from the schedule	$O(\log(n))$	Not supported

Table 1: Comparison of Θ-tree and time line.

The data structure is inspired from López-Ortiz et al. [16]. We consider a sequence $t[0..|t|-1]$ of unique time points, sorted in chronological order, formed by the earliest starting times of the tasks, and a sufficiently large time point, which is $\max_{i \in \mathcal{I}} \text{lct}_i + \sum_{i=1}^{n} p_i$. The vector $m[0..n-1]$ maps a task i to the time point index such that $t[m[i]] = \text{est}_i$. The time points, except for the last one, have a residual capacity, stored in the vector $c[0..|t|-2]$. The residual capacity $c[a]$ denotes the amount of time the resource is available within the semi-open time interval $[t[a], t[a+1])$ should the tasks in Θ be scheduled at their earliest starting time with preemption. Initially, since $\Theta = \emptyset$, the resource is fully available and $c[a] = t[a+1] - t[a]$. A Union-Find data structure s is initialized with $|t|$ elements. This data structure maintains the invariant that a and $a+1$ belong

to the same set in s if and only if $c[a] = 0$. This allows us to quickly request, by invoking s.`FindGreatest`(a), the earliest time point no earlier than $t[a]$ with a positive residual capacity. Finally, the data structure has an index e which is the index of the latest time point whose residual capacity has been decremented. Algorithm 1 initializes the time line data structure. It proceeds by initializing the components t, m, c, s, and e that define the time line data structure.

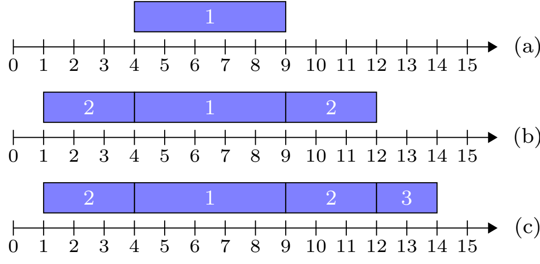
Algorithm 1: `InitializeTimeline`$(\mathcal{I})$

```
t ← [], c ← [];
for i ∈ I_est do
    if |t| = 0 ∨ t[|t| − 1] ≠ est_i then
        t.append(est_i);
    m[i] ← |t| − 1;
t.append(max_i lct_i + Σ_{i=1}^{n} p_i);
for k = 0..|t| − 2 do
    c[k] ← t[k + 1] − t[k];
s ← UnionFind(|t|);
e ← −1;
```

The data structure allows to "schedule" a task i on the time line at its earliest time and with preemption. The value $m[i]$ maps the task i to the index of the time point associated to the earliest starting time of task i. Algorithm 2 schedules a task on the time line. It iterates through the time intervals $[t[m[i]], t[m[i]+1]), [t[m[i]+1], t[m[i]+2]), \ldots$ and decreases the residual capacity of each interval down to 0 until a total of p_i units of capacity is decreased. Each time a residual capacity $c[k]$ reaches zero, the union-find merges the index k with $k+1$ which allows, in the future, to skip arbitrarily long sequences of intervals with null residual capacities in constant time. The function `ScheduleTask` returns the time at which the newly scheduled task on the time line completed.

Algorithm 2: `ScheduleTask`(i)

```
ρ ← p_i;
k ← s.FindGreatest(m[i]);
while ρ > 0 do
    Δ ← min(c[k], ρ);
    ρ ← ρ − Δ;
    c[k] ← c[k] − Δ;
    if c[k] = 0 then
        s.Union(k, k + 1);
        k ← s.FindGreatest(k);
e ← max(e, k);
return t[k + 1] − c[k]
```



i	est_i	lct_i	p_i
1	4	15	5
2	1	10	6
3	5	8	2

Fig. 5: A trace of Example 5.

Let Θ be the tasks that were scheduled on the time line using Algorithm 2. Algorithm 3 computes ect_Θ in constant time.

Example 5 Consider the tasks whose parameters are stated on Figure 5. Initializing the time line produces the structure

$$\{1\} \overset{3}{\to} \{4\} \overset{1}{\to} \{5\} \overset{23}{\to} \{28\}$$

where the numbers in the sets are the time points and the numbers on the arrows denote the residual capacities. After executing `ScheduleTask`(1) (see Figure 5a), the residual capacity between the time points 4 and 5 becomes null and the union-find merges both time points into the same set. The structure becomes $\{1\} \overset{3}{\to} \{4, 5\} \overset{19}{\to} \{28\}$. After calling `ScheduleTask`(2) (Figure 5b), the time line becomes $\{1, 4, 5\} \overset{16}{\to} \{28\}$ and after calling `ScheduleTask`(3) (Figure 5c), it becomes $\{1, 4, 5\} \overset{14}{\to} \{28\}$. The function `EarliestCompletionTime` retrieves the earliest completion time by computing $28 - 14 = 14$.

Theorem 1 *Algorithm 1 runs in $O(n)$ amortized time while Algorithm 2 and Algorithm 3 run in constant amortized time.*

Proof Let c_i be the residual capacity vector after the ith call to an algorithm among Algorithm 1, Algorithm 2, and Algorithm 3. We define a potential function $\phi(i) = |\{k \in 0..|t|-2 \mid c_i[k] > 0\}|$ that is equal to the number of positive components in the vector c_i. Prior to the initialization of the time line data structure, we have $\phi(0) = 0$, as the residual capacity vector is empty. Always we have $\phi(i) \geq 0$. After the initialization, we have $\phi(1) = |t| - 1 \leq n$. The two for loops in Algorithm 1 execute $n + |t| - 1 \leq 2n \in O(n)$ times. Therefore, the amortized complexity of the initialization is $O(n) + \phi(1) - \phi(0) = O(n)$.

Suppose the while loop in Algorithm 2 executes a times. There are at least $a - 1$ and at most a components in the residual capacity vector that are set to

Algorithm 3: `EarliestCompletionTime`(e)

```
if e ≥ 0 then
|  t[e + 1] - c[e]
else
⌊ return −∞;
```

zero, hence $a - 1 \leq \phi(i) - \phi(i-1) \leq a$. The amortized complexity of Algorithm 2 is therefore $a + \phi(i) - \phi(i-1) \leq a - (a-1) \in O(1)$.

Algorithm 3 executes in constant time and does not modify the residual capacity vector c which implies $\phi(i) = \phi(i-1)$. The amortized complexity is therefore $O(1) + \phi(i) - \phi(i-1) = O(1)$. □

5 Novel Algorithms

5.1 Time-Tabling

We present Algorithm 4, a linear time algorithm that enforces the Time-Tabling rule on the Disjunctive constraint. For this algorithm, the union-find data structure is sufficient to implement the algorithm and therefore it does not require the use of the time line data structure. This new algorithm is not idempotent. However, it provides some guarantees on the level of filtering it achieves. Consider the set of compulsory parts $\mathcal{F} = \{[\text{lst}_i, \text{ect}_i) \mid i \in \mathcal{I} \land \text{lst}_i < \text{ect}_i\}$. Consider a task $j \in \mathcal{I}$. The algorithm guarantees that after the filtering occurs, the interval $[\text{est}'_j, \text{ect}'_j)$ does not intersect with any interval in $\mathcal{F}$. However, the pruning of est_j to est'_j might create a new compulsory part $[\text{lst}_j, \text{ect}'_j)$ that could cause some filtering in a further execution of the algorithm.

These guarantees are identical to those that the Sweep algorithm in [3] offers. Letort et al. [15] improve the Sweep algorithm to update the set of compulsory parts $\mathcal{F}$ with the newly created compulsory part $[\text{lst}_j, \text{ect}'_j)$ leading to fewer iterations before converging to the fixed point. In contrast, Gay et al. [10] propose an algorithm which filters less than the one we propose, but it achieves a linear time complexity. They also propose a quadratic-time algorithm that does the same level of filtering as the Sweep algorithm by Letort et al. [15]. All algorithms that achieve Time-Tabling converge to a fixed point in a polynomial number of iterations. It is more likely that the algorithms that filter more in a single iteration converge faster to the fixed point.

Algorithm 4 proceeds in three steps, each of which is associated to a for loop. The first for loop on line 1 creates the vectors l and u that contain the lower bounds and upper bounds of the compulsory parts. The compulsory parts $[l[0], u[0])$, $[l[1], u[1])$, $\ldots$, $[l[m-1], u[m-1])$ form a sequence of sorted and disjoint semi-open intervals such that each of them is associated to a task i that satisfies $\text{lst}_i < \text{ect}_i$. If two compulsory parts overlap, on line 2 the algorithm returns *Inconsistent*. When processing the task i that has a compulsory part $[l[k], u[k])$, on line 3 the algorithm ensures that the task i starts no earlier than $u[k-1]$, so that the tasks that have a compulsory part are all filtered.

The second for loop on line 4 creates a vector r that maps a task i to the compulsory part whose upper bound is the smallest one to be greater than est_i. Therefore, the relation $u[r[i]-1] \leq \text{est}_i < u[r[i]]$ holds.

The third for loop on line 5 filters the tasks that do not have a compulsory part. The tasks are processed by non-decreasing order of processing times. Line 6 checks whether $\text{est}'_i + p_i > l[r[i]]$. If so, then the Time-Tabling rule applies and the new value of est'_i is pruned to $u[c]$. The same task is then checked against the next compulsory part $[l[r[i]+1], u[r[i]+1])$ and so on. Suppose that a task is filtered both by the compulsory part $[l[c], u[c])$ and the compulsory part $[l[c+1], u[c+1])$.

Algorithm 4: TimeTabling($\mathcal{I}$)

```
   m ← 0, k ← 0, l ← [], u ← [], r ← [];
   est'_i ← est_i, ∀ i ∈ I;
1  for i ∈ I_lst do
       if lst_i < ect_i then              // If the task i has a compulsory part
           if m > 0 then
2              if u[m − 1] > lst_i then  return Inconsistent;
3              est'_i ← max(est'_i, u[m − 1]);
           l.append(lst_i);
           u.append(est'_i + p_i);
           m ← m + 1;
   if m = 0 then      // Without compulsory parts, no filtering is needed
       return Consistent;
4  for i ∈ I_est do
       while k < m ∧ est_i ≥ u[k] do
           k ← k + 1;
       r[i] ← k;
   s ← UnionFind(m);
5  for i ∈ I_p do
       if ect_i ≤ lst_i then
           c ← r[i];
           first_update ← True;
6          while c < m ∧ est'_i + p_i > l[c] do
               c ← s.FindGreatest(c);
               est'_i ← max(est'_i, u[c]);
               if est'_i + p_i > lct_i then return Inconsistent;
               if ¬first_update then
                   s.Union(r[i], c);
               first_update ← False;
               c ← c + 1;
   return Consistent;
```

Since we process the tasks by non-decreasing order of processing times, any further task that is filtered by the compulsory part $[l[c], u[c])$ will also be filtered by the compulsory part $[l[c+1], u[c+1])$. The algorithm uses a Union-Find data structure to ensure that these two compulsory parts are *glued* together. The next task j that satisfies $\text{est}'_j + p_j > l[c]$ will be filtered to $u[c+1]$ in a single iteration. The Union-Find data structure can union an arbitrary long sequence of compulsory parts.

Theorem 2 *Algorithm 4 enforces the Time-Tabling rule in $O(n)$ steps.*

Proof Each of the two first for loops iterate through the tasks once and execute operations in constant time. Each time the while loop on line 6 executes more than once, the Union-Find data structure merges two compulsory parts. This can occur at most n times. □

5.2 Overload Checking

The Overload Checking algorithm, as described by Vilím [29], can be directly implemented with a time line data structure rather than a Θ-tree. One schedules the tasks on the time line, with preemption, using Algorithm 2, in non-decreasing order of latest completion times. If after scheduling a task i on the time line, Algorithm 3 returns an earliest completion time greater than lct_i, then the Overload Checking fails. The total running time complexity of this algorithm is $O(n)$. The proof of correctness follows Vilím's.

The Overload Checking can be adapted to the CUMULATIVE constraint with a resource of capacity $\mathcal{C}$. Following Section 2, one transforms the task i of height h_i into a task i' with $\mathrm{est}_{i'} = \mathcal{C} \cdot \mathrm{est}_i$, $\mathrm{lct}_{i'} = \mathcal{C} \cdot \mathrm{lct}_i$, $p_{i'} = \mathrm{e}_i$, and $h_{i'} = 1$. The Overload Checking fails on the original problem if and on if it fails on the transformed model. The transformation preserves the running time complexity of $O(n)$.

5.3 Not-first/Not-last

Kameugne et al. [14] introduce a sound Not-first/Not-last algorithm running in $O(n^2 \log n)$ time (see Algorithm 5), which turns out to be the state-of-the-art for the CUMULATIVE constraint.

Algorithm 5: `NotFirst`($\mathcal{I}$) (Kameugne et al. [14])

```
    for i ∈ I do
        est'_i ← est_i;
    for i ∈ I do
1       Θ ← ∅;
        minEct ← ∞;
        for j ∈ I do
            if est_i < ect_j ∧ j ≠ i then
2               Θ ← Θ ∪ {j};
                minEct ← min(minEct, ect_j);
3               if Env(Θ, h_i) > C · lct_j − h_i · min(ect_i, lct_j) then
                    est'_i ← max(est'_i, minEct);
                    break;
```

Algorithm 5 uses a Θ-tree to compute the energy envelope of Θ with respect to a task i at line 3. The energy envelope $Env(\Theta, h_i)$ is computed by choosing a subset of tasks $\Omega \subseteq \Theta$ that maximizes the amount of energy required to consume $\mathcal{C} - h_i$ units of resource over the time interval $[0, \mathrm{est}_\Omega)$ in addition to the energy of the tasks in Ω.

$$Env(\Theta, h_i) = \max_{\Omega \subseteq \Theta} (\mathcal{C} - h_i)\, \mathrm{est}_\Omega + \mathrm{e}_\Omega$$

Using the transformation from Section 2, one can compute the energy envelope $Env(\Theta, h_i)$ using the time line. One transforms the tasks in $\mathcal{I}$ into a set of tasks $\mathcal{I}'$.

The task $j \in \mathcal{I}$ becomes a task $j' \in \mathcal{I}'$ with $\text{est}_{j'} = (\mathcal{C} - h_i) \cdot \text{est}_j$, $\text{lct}_{j'} = (C - h_i) \cdot \text{lct}_j$, $p_{j'} = \text{e}_j$, and $h_{j'} = 1$. The line 1 of Algorithm 5 initializes a time line using the set of tasks $\mathcal{I}'$ by calling Algorithm 1. Algorithm 2 is used to schedule task j' on the time line at line 2. From Lemma 1, we obtain that $Env(\Theta, h_i)$ is given by the earliest completion time of the time line. In other words, Algorithm 3 returns $Env(\Theta, h_i)$.

The energy envelope computed by the time line is always the same as the energy envelope computed by the Θ-tree. Therefore, the time line yields the same filtering as the Θ-tree. The time line removes a $\log n$ factor from the running time of the algorithm, leading to a total running time complexity of $O(n^2)$. The proof of correctness follows the one by Kameugne et al [14].

5.4 Detectable Precedences

We introduce a new algorithm to enforce the rule of detectable precedences for the Disjunctive constraint. The algorithm by Vilím et al. [34] cannot be simply adapted for the time line data structure, as it requires to temporarily remove a task from the current schedule. As mentionned on Table 1, this operation is not supported by the time line. Nonetheless, our method, presented in Algorithm 6, circumvents this issue while maintaining linearity for the complexity of the algorithm.

Suppose that the problem has no tasks with a compulsory part, i.e. $\text{ect}_i \leq \text{lst}_i$ for all task $i \in \mathcal{I}$. The algorithm simultaneously iterates over all the tasks i in non-decreasing order of earliest completion times and on all the tasks k in non-decreasing order of latest starting times. Each time the algorithm iterates over the next task i, it iterates (line 2) and schedules on the time line (line 3) all tasks k whose latest starting time lst_k is smaller than the earliest completion time ect_i. Once the while loop completes, the set of tasks scheduled on the time line is $\{k \in \mathcal{I} \setminus \{i\} \mid \text{lst}_k < \text{ect}_i\}$. We apply the detectable precedence rule by pruning the earliest starting time of task i up to the earliest completion time of the time line (line 4).

Suppose that there exists a task k with a compulsory part, i.e. $\text{ect}_k > \text{lst}_k$. This task could be visited in the while loop before being visited in the main for loop. We do not want to schedule on the time line the task k before it is filtered. We therefore call the task k the *blocking task*. When a blocking task k is encountered in the while loop, the algorithm waits to encounter the same task in the for loop. During this waiting period, the filtering of all tasks is postponed. A postponed task i necessarily satisfies the conditions $\text{lst}_k < \text{ect}_i \leq \text{ect}_k$ and $\text{ect}_i < \text{lst}_i$. Since $\text{lst}_k < \text{ect}_i$ then the precedence $k \ll i$ holds. When the for loop reaches the blocking task k, it filters the blocking task, schedules on the time line the blocking task, and filters the postponed tasks. The blocking task and the set of postponed tasks are reset. It is not possible to simultaneously have two blocking tasks since their compulsory parts would overlap, which is inconsistent with the Time-Tabling rule.

Example 6 Figure 6 shows a trace of Algorithm 6. The for loop on line 1 processes the tasks $\mathcal{I}_{\text{ect}} = \{1, 2, 3, 4\}$ in that order. For the two first tasks 1 and 2, nothing happens: the while loop is not executed and no pruning occurs as no tasks are scheduled on the time line. When the for loop processes task 3, the while loop

Algorithm 6: DetectablePrecedences($\mathcal{I}$)

```
  InitializeTimeline (I);
  j ← 0;
  k ← I_lst[j];
  postponed_tasks ← ∅;
  blocking_task ← null;
1 for i ∈ I_ect do
2     while j < |I| ∧ lst_k < ect_i do
          if lst_k ≥ ect_k then
3             ScheduleTask (k);
          else
              if blocking_task ≠ null then return Inconsistent;
              blocking_task ← k;
          j ← j + 1;
          k ← I_lst[j];
      if blocking_task = null then
4         est'_i ← max(est_i, EarliestCompletionTime());
      else
          if blocking_task = i then
              est'_i ← max(est_i, EarliestCompletionTime());
              ScheduleTask (blocking_task);
              for z ∈ postponed_tasks do
5                 est'_z ← max(est_z, EarliestCompletionTime());
              blocking_task ← null;
              postponed_tasks ← ∅;
          else
              postponed_tasks ← postponed_tasks ∪ {i};
  for i ∈ I do
      est_i ← est'_i;
```

processes three tasks. The while loop processes the task 2 which is scheduled on the time line. When it processes task 4, the while loop detects that task 4 has a compulsory part in $[14, 18)$ making task 4 the blocking task. Finally, the while loop processes task 1 which is scheduled on the time line. Once the while loop completes, the task 3 is not filtered since there exists a blocking task. Its filtering is postponed until the blocking task is processed. The for loop processes the task 4. In this iteration, the while loop does not execute. Since task 4 is the blocking task, it is first filtered to the earliest completion time computed by the time line data structure ($\text{est}'_4 \leftarrow 13$). Task 4 is then scheduled on the time line. Finally, the postponed task 3 is filtered to the earliest completion time computed by the time line data structure ($\text{est}'_3 \leftarrow 19$).

Theorem 3 *The algorithm* DetectablePrecedences *runs in linear time.*

Proof The for loop on line 1 processes each task only once, idem for the while loop. Finally, a task can be postponed only once during the execution of the algorithm

i	est_i	lct_i	p_i	ect_i	lst_i	postponed_tasks	est'_i
1	0	19	4	4	15	$\emptyset$	0
2	2	22	9	11	13	$\emptyset$	2
3	9	30	7	16	23	$\{3\}$	19
4	12	20	6	18	14	$\emptyset$	13

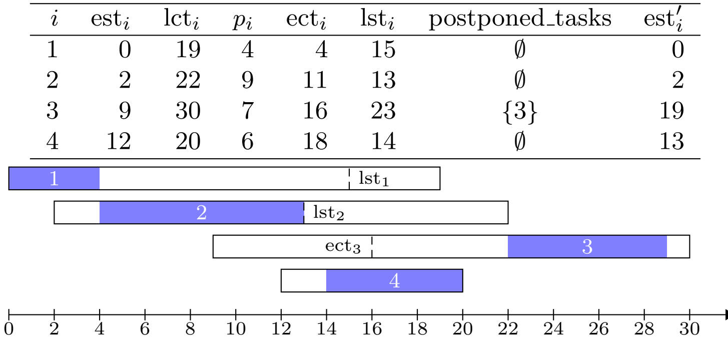


Fig. 6: The tasks $\mathcal{I}_{ect} = \{1, 2, 3, 4\}$ and the visual representation of a solution to the Disjunctive constraint. The algorithm `DetectablePrecedences` prunes the earliest starting times $\text{est}'_3 = 19$ and $\text{est}'_4 = 13$.

and therefore line 5 is executed at most n times. Except for `InitializeTimeline` and the sorting of $\mathcal{I}_{\text{ect}}$ and $\mathcal{I}_{\text{lst}}$ that are executed once in $O(n)$ time, all other operations execute in amortized constant time. Therefore, `DetectablePrecedences` runs in linear time. □

5.5 Minimizing maximum lateness and total delay

Numerous objective functions can be optimized in a scheduling problem. Let E_i denote the completion time of a task i. The *due date* of a task, denoted d_i, is the last time that a task is expected to complete, without incurring penalties. The *lateness* and *delay* of a task i are computed with $L_i = E_i - d_i$ and $D_i = E_i - \text{est}_i$. We consider two problems: minimizing the maximum lateness ($\max_{i \in \mathcal{I}} L_i$) and minimizing total delay ($\sum_{i \in \mathcal{I}} D_i$).

Not only is the time line data structure useful to design filtering algorithms for global scheduling constraints, but also it is strong enough to efficiently solve simple preemptive scheduling problems whose best known algorithm runs in $\Theta(n \log(n))$. While these problems are generally easy to solve, improving their running time complexity is still relevant. Indeed, these simple problems can be used as a relaxation for more complex problems and the algorithms we present can be used within a branch and bound.

The first algorithms for the problems of minimizing maximum lateness and minimizing total delay in the disjunctive and *preemptive* case were introduced in [13]. These algorithms are roughly similar. To minimize maximum lateness, the algorithm in [13] schedules the tasks, with preemption and at their earliest time, in non-decreasing order of due dates. To minimize the total delay, the algorithm schedules the tasks in non-decreasing order of processing times.

It turns out that the time line provides a more efficient way to implement these algorithms as one can see with Algorithms 7 and 8. These algorithms use the ending time of the tasks returned by the function `ScheduleTask` to compute the optimal objective value.

Algorithm 7: `MinimizingMaximumLateness`($\mathcal{I}$)

`InitializeTimeline` ($\mathcal{I}$);
for $i \in \mathcal{I}_d$ **do**
  $E_i \leftarrow$ `ScheduleTask`(i);
  $L_i \leftarrow E_i - d_i$;
$L_{\max} \leftarrow \max_{i \in \mathcal{I}}(L_i)$;
return $L_{\max}$;

Algorithm 8: `MinimizingTotalDelay`($\mathcal{I}$)

`InitializeTimeline` ($\mathcal{I}$);
for $i \in \mathcal{I}_p$ **do**
  $E_i \leftarrow$ `ScheduleTask`(i);
  $D_i \leftarrow E_i - \text{est}_i$;
$D \leftarrow \sum_{i \in \mathcal{I}} D_i$;
return D;

citevilim2004nlog Horn [13] proves that these algorithms minimize the maximum lateness and total delays. Since `ScheduleTask` runs in constant time, the overall complexities of Algorithms 7 and 8 are linear.

6 Experimental Results

We conducted two experiments. The first experiment compares the filtering algorithm for the Disjunctive constraint against the state-of-the-art algorithms that perform the same level of filtering. The second experiment performs a similar comparison for the Cumulative constraint.

We implemented our algorithms in Choco 2.1.5 and, as a point of comparison, the Overload Checking and the Detectable Precedences from Vilím [29], the Time Tabling algorithm from Ouellet and Quimper [20], and the Not-first/Not-last from Kameugne et al. [14]. All experiments were run on an Intel Xeon X5560 2.667GHz quad-core processor. We used the impact based search and *DomOverWDeg* heuristic with a timeout of 10 minutes. These search strategies were already available in the solver and were sufficient to compare the execution times of the algorithms. However, there exist other heuristics, not implemented in that specific version of Choco, that could be more suited for solving scheduling problems. Unless specified otherwise, each filtering algorithm is individually tested, i.e. we did not combine the filtering algorithms. For the instances that were solved to optimality within 10 minutes, the two filtering algorithms of the same technique, whether it is Overload Checking, Detectable Precedences, Time-Tabling, or Not-first/Not-last produce the same number of backtracks since they achieve the same filtering and therefore explore the same search tree.

$n \times m$	OC	DP	TT
4×4	0.99	1.00	1.00
5×5	1.03	1.00	1.72
7×7	0.99	1.08	1.74
10×10	1.18	1.33	2.31
15×15	1.06	1.22	2.58
20×20	1.82	1.41	2.91

Table 2: Open-shop with n jobs and m tasks per job. Ratio of the average number of backtracks between all instances of size $n \times m$ after 10 minutes of computations. OC: our Overload Checking vs. Vilím's. DP: our Detectable Precedences vs Vilím's. TT: Our Time-Tabling vs Ouellet et al.

6.1 Algorithms for the Disjunctive constraint

We compare our filtering algorithms against the state-of-the-art filtering algorithms enforcing the same level of filtering. For all algorithms, we sort the tasks using the insertion sort, which mainly keeps track of the vector which was sorted by the previous call to the algorithm and reuses that in the current iteration.

The experiments are carried out on the job-shop and open-shop scheduling problems. In these problems, n jobs, consisting of a set of non-preemptive tasks, execute on m machines. Each task executes on a predetermined machine with a given processing time. In the job-shop problem, the tasks belonging to the same job execute in a predetermined order. In the open-shop problem, the number of tasks per job is fixed to m and the order in which the tasks of a job are processed is immaterial. In both problems, the goal is to minimize the makespan. We use the benchmark provided by [27] that includes 82 and 60 instances of the job-shop and open-shop problems.

We model the problems with a starting time variable $S_{i,j}$ for each task j of job i. We post a Disjunctive constraint over the starting time variables of tasks running on the same machine. For the job-shop scheduling problem, we add the precedence constraints $S_{i,j} + p_{i,j} \leq S_{i,j+1}$. For the open-shop scheduling problem, we add a Disjunctive constraint among all tasks of a job. We declare a makespan variable $E_{\max}$ subject to the constraints $E_{\max} \geq S_{i,j} + p_{i,j}$. For the job-shop problem, this last inequality is only posted on the last task of each job.

To compare the algorithms, for all instances of the same size, we average the number of backtracks achieved within 10 minutes and we report the ratio of these backtracks between both algorithms. Since both algorithms explore precisely the same search tree in the same order, a ratio greater than 1 indicates that our algorithms explore a larger portion of the search tree and therefore they perform faster.

Tables 2 and 3 show the results for the open-shop and job-shop problems. From these tables, we cannot conclude that the new Overload Checking is faster and we cannot conclude that it is slower, either. This is caused by the initialization of the time line that is slower than the initialization of the $\Theta-$tree. In order to amortize the initialization time, the Overload Checking requires to process instances larger than those in Table 2 and 3.

$n \times m$	OC	DP	TT
10×5	0.94	1.12	1.86
15×5	0.95	1.16	2.07
20×5	0.94	1.37	2.15
10×10	0.95	1.13	2.10
15×10	0.84	1.20	2.06
20×10	0.93	1.34	2.48
30×10	0.95	1.38	2.80
50×10	1.02	1.51	3.29
15×15	0.90	1.14	2.38
20×15	0.89	1.38	2.35
20×20	0.92	1.25	1.70

Table 3: Job-shop with n jobs and m tasks per job. Ratio of the average number of backtracks between all instances of size $n \times m$ after 10 minutes of computations. OC: our Overload Checking vs. Vilím's. DP: our Detectable Precedences vs Vilím's. TT: Our Time-Tabling vs Ouellet et al.

The new algorithm of Detectable Precedences shows improvements on both problems especially when the number of variables increases. One way to explain why the ratios are greater compared with the Overload Checking is that the most costly operations in Vilím's algorithm is the insertion and removal of a task in the Θ-tree which can occur up to 3 times for each task. With the new algorithm, the most costly operation is the scheduling of a task on the time line which occurs only once per task.

The ratios verify that the new Time-Tabling algorithm is faster for both job shop and open shop problems. The new Time-Tabling is much faster than the one by Ouellet and Quimper [20], probably because our time tabling is specialised for the Disjunctive constraint while Ouellet's one is designed for the Cumulative constraint, a more general constraint.

Furthermore, we randomly generated large but easy instances with a single Disjunctive constraint over variables with uniformly generated domains. Unsatisfiable instances and instances solved with zero backtracks were discarded. Table 4 shows the ratio for the number of branches per millisecond and it verifies that the new algorithms are consistently faster, including the new overload checking when used on larger instances.

We also combined the filtering algorithms together to see how well they interact together. We tried the combinations (OC+DP), (OC+TT), (DP+TT) and (OC+DP+TT) with our proposed algorithms. Few instances were solved to optimality, but the quality of the makespan obtained (smaller is better) allows us to compare the performance of the different combinations. For the job shop problem, on 55% of the instances (OC+DP) yields the best makespan, on 17% of the instances (OC+TT) yields the best makespan, on 27% of the instances (DP+TT) yields the best makespan and on 1% of the instances (OC+DP+TT) yields the best makespan. For the open shop problem, on 55% of the instances (OC+DP) yields the best makespan, on 22% of the instances (OC+TT) yields the best makespan and on 23% of the instances (DP+TT) yields the best makespan. Thus, Overload

	Overload Checking		Detectable Precedences		Time-Tabling	
n	TT (bt/ms)	TL (bt/ms)	TT (bt/ms)	TL (bt/ms)	Ouellet (bt/ms)	UF (bt/ms)
10	**261.02**	248.3	10.29	**16.11**	56.62	**122.31**
20	154.17	**168.97**	32.04	**48.34**	30.65	**104.46**
30	**137.30**	137.03	28.09	**43.15**	46.67	**133.89**
40	**52.12**	49.28	37.71	**60.81**	32.14	**83.16**
50	27.00	**27.06**	51.69	**92.55**	16.17	**69.54**
60	57.01	**57.68**	23.59	**41.72**	12.95	**60.00**
70	31.38	**32.69**	11.41	**20.24**	17.43	**78.76**
80	20.13	**20.20**	31.33	**46.96**	17.60	**45.01**
90	50.23	**51.09**	23.57	**39.33**	24.41	**66.88**
100	17.97	**19.22**	24.78	**42.2**	21.76	**83.91**

Table 4: Random instances with n tasks. The numbers in bold denote which data structure performs faster. Times are measured in milliseconds. bt/ms stands for the number of branches per millisecond. TT: $\Theta-$tree, TL: time line, UF:Union-Find data structure.

Checking combined with Detectable Precedences is a strong combination to solve both open shop and job shops problems.

6.2 Algorithms for the Cumulative constraint

Experiments for the Cumulative constraint were conducted on the PSPLIB benchmarks [1] of the Resource Constrained Project Scheduling Problems (RCPSP). This problem consists of multiple resources of different capacities on which n tasks can simultaneously be executed. Each task i has a set of predecessors. The goal is to minimize the makespan.

We model the problem with a starting time variable S_i for each task i. For each resource, we post a Cumulative constraint over the starting time variables of all tasks using that resource. Precedence constraint between two tasks i and j are modeled with the constraint $S_i + p_i \leq S_j$. We declare a makespan variable $E_{\max}$ subject to the constraints $E_{\max} = \max(S_1 + p_1, \ldots, S_n + p_n)$ that we minimize. We implemented in Choco 2.1.5 the Not-first/Not-last algorithm from [14] with the time line and, as a point of comparison, with the Θ-tree. We did not use other filtering algorithms. We used the heuristic *DomOverWDeg*.

On the 2040 instances of the PSPLIB benchmark, 395 were optimally solved within ten minutes by both versions of the algorithm. The solved instances are distributed among the 30-task instances (154), the 60-task instances (121), and the 90-task instances (120). The majority of these instances were solved under one second, but 18 of the 30-task instances were more difficult. They were solved with an average of 80.05 seconds for the Θ-tree and 51.94 seconds for the time line. None of the 120-task instances were solved to optimality. Figure 7 shows that the time line does significantly more backtacks per second than the Θ-tree for almost all instances. The time line is therefore faster than the Θ-tree.

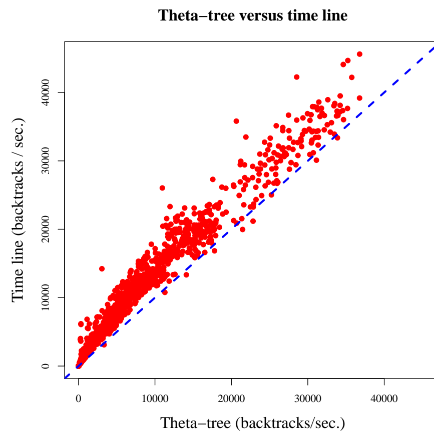


Fig. 7: Comparison of the backtracks per seconds of the Not-first/Not-last algorithm with the Θ-tree versus the same algorithm with time line for instances of the PSPLIB benchmark with a time limit of ten minutes.

6.3 Further Discussion

The experimental results confirm that the time line data structure can improve the solving time of the solver. While we now have faster algorithms for the Time-Tabling, Overload Checking, Not-First/ Not-Last, and Detectable Precedences, it is important to point out that our experimental setup did not outperform state of the art methods for solving the RCPSP instances. Indeed, a careful combination of branching heuristics, search strategies, filtering algorithms, and also lazy clause generation is necessary to achieve good performances [23, 24]. This strongly motivates to further exploit the time line data structure not only to filter, but also to explain the filtering.

The Time-Table Edge-Finding [20, 23, 33] filtering rule offers a good compromise between the filtering level and the computation time. It remains an open question whether the time line can be used to improve the computation time required to apply this rule.

7 Conclusion

We introduced a new data structure called the *time line*. We took advantage of this data structure to present three new filtering algorithms for the Disjunctive constraint that all have a linear running time complexity in the number of tasks. For the Cumulative constraint, our linear-time Overload Checking can be adapted and the time line provides an improvement for Not-first/Not-last. The new algorithms outperform the best algorithms known so far that achieve the same level of filtering. This data structure can also be exploited to achieve more efficient algorithms to solve the scheduling problems of minimizing maximum lateness and total delay.

As future work, we believe that the time line data structures could be adapted to provide explanations and therefore allow lazy clause generation. It also remains an open question whether the Edge-Finding and the Time-Table-Edge-Finding can be adapted to use the time line data structure to apply the rules in $O(kn)$ time where k is the number of distinct task heights.

References

1. Baptiste, P., Le Pape, C.: Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. Constraints **5**(1-2), 119–139 (2000)
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: Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002), pp. 63–79 (2002)
4. Beldiceanu, N., Carlsson, M., Poder, E.: New filtering for the cumulative constraint in the context of non-overlapping rectangles. In: Proceedings of the 5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR 2008), pp. 21–35 (2008)
5. Belov, G., Boland, N., Savelsbergh, M.W.P., Stuckey, P.J.: Exploration of models for a cargo assembly planning problem. ArXiv e-prints (2015)
6. Caseau, Y., Laburthe, F.: Cumulative scheduling with task intervals. In: Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP), pp. 369–383 (1996)
7. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms, 2nd edn. McGraw-Hill Higher Education (2001)
8. Fahimi, H., Quimper, C.G.: Linear-time filtering algorithms for the disjunctive constraint. In: AAAI, pp. 2637–2643 (2014)
9. Gabow, H.N., Tarjan, R.E.: A linear-time algorithm for a special case of disjoint set union. In: Proceedings of the 15th annual ACM symposium on Theory of computing, pp. 246–251 (1983)
10. Gay, S., Hartert, R., Schaus, P.: Simple and scalable time-table filtering for the cumulative constraint. In: Principles and Practice of Constraint Programming, pp. 149–157. Springer (2015)
11. Gay, S., Schaus, P., Smedt, V.D.: Continuous casting scheduling with constraint programming. In: International conference on principles and practice of constraint programming, pp. 831–845. Springer (2014)
12. Guéret, C., Jussien, N., Boizumault, P., Prins, C.: Building university timetables using constraint logic programming. In: International Conference on the Practice and Theory of Automated Timetabling, pp. 130–145. Springer (1995)
13. Horn, W.: Some simple scheduling algorithms. Naval Research Logistics Quarterly **21**(1), 177–185 (1974)
14. Kameugne, R., Fotso, L.P.: A cumulative not-first/not-last filtering algorithm in o (n 2log (n)). Indian Journal of Pure and Applied Mathematics **44**(1), 95–115 (2013)
15. Letort, A., Beldiceanu, N., Carlsson, M.: A scalable sweep algorithm for the cumulative constraint. In: Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP 2012), pp. 439–454 (2012)
16. López-Ortiz, A., Quimper, C.G., Tromp, J., van Beek, P.: A fast and simple algorithm for bounds consistency of the alldifferent constraint. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03), pp. 245–250 (2003)
17. Le Pape, C.: Des systèmes d'ordonnancement flexibles et opportunistes. Ph.D. thesis, Universit Paris IX (1988)
18. Mehlhorn, K., Thiel, S.: Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. CP **2**, 306–319 (2000)
19. Nuijten, W.W.: Time and resource constrained scheduling: a constraint satisfaction approach. Ph.D. thesis, Technische Universiteit Eindhoven (1994)

20. Ouellet, P., Quimper, C.G.: Time-table-extended-edge-finding for the cumulative constraint. In: Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP 2013), pp. 562–577 (2013)
21. Puget, J.F.: A fast algorithm for the bound consistency of alldiff constraints. In: Proceedings of the 15th National Conference on Artificiel Intelligence (AAAI-98) and the 10th Conference on Innovation Applications of Artificial Intelligence (IAAI-98), pp. 359–366 (1998)
22. Quimper, C.G., López-Ortiz, A., Pesant, G.: A quadratic propagator for the inter-distance constraint. In: Proc. of the 21st Nat. Conf. on Artificial Intelligence (AAAI 06), pp. 123–128 (2006)
23. Schutt, A., Feydy, T., Stuckey, P.J.: Explaining time-table-edge-finding propagation for the cumulative resource constraint. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2013), pp. 234–250 (2013)
24. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Solving rcpsp/max by lazy clause generation. Journal of Scheduling **16**(3), 273289 (2013)
25. Schutt, A., Stuckey, P., Verden, A.: Optimal carpet cutting. pp. 69–84 (2011)
26. Schutt, A., Wolf, A., Schrader, G.: Not-first and not-last detection for cumulative scheduling in $O(n^3 \log(n))$. In: International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2005), pp. 66–80 (2006)
27. Taillard, E.: Benchmarks for basic scheduling problems. European Journal Opererational Research **64**(2), 278–285 (1993)
28. Vilím, P.: Batch processing with sequence dependent setup times: New results. In: Proceedings of the 4th Workshop of Constraint Programming for Decision and Control (CPDC'02) (2002)
29. Vilím, P.: $o(n \log n)$ filtering algorithms for unary resource constraint. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pp. 335–347. Springer (2004)
30. Vilím, P.: Global constraints in scheduling. Ph.D. thesis, Charles University in Prague (2007)
31. Vilím, P.: Edge finding filtering algorithm for discrete cumulative resources in $o(knlogn)$. In: International Conference on Principles and Practice of Constraint Programming, pp. 802–816. Springer (2009)
32. Vilím, P.: Max energy filtering algorithm for discrete cumulative resources. In: Integration of AI and OR techniques in constraint programming for combinatorial optimization problems, pp. 294–308. Springer (2009)
33. Vilím, P.: Timetable edge finding filtering algorithm for discrete cumulative resources. In: Proceedings of the 8th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2011), pp. 230–245 (2011)
34. Vilím, P., Barták, R., Čepek, O.: Unary resource constraint with optional activities. In: Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming, pp. 62–76 (2004)
35. Wolf, A., Schrader, G.: $o(n \log(n))$ overload checking for the cumulative constraint and its application. In: International Conference on Applications of Declarative Programming and Knowledge Management, pp. 88–101. Springer (2005)