Explaining detectable precedences for the disjunctive constraint

Matthias van Vliet[1]
Supervisors: Emir Demirović[1]**, Imko Marijnissen**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Matthias van Vliet
Final project course: CSE3000 Research Project
Thesis committee: Emir Demirović, Imko Marijnissen, Stephanie Wehner

Explaining detectable precedences for the disjunctive constraint

Matthias van Vliet
Delft University of Technology, The Netherlands

Abstract

As lazy clause generation has seen much success in recent years, the generation of explanations has become the focus of much research. This paper describes how explanations can be generated for detectable precedences in the disjunctive constraint. We also provide a method to incorporate these explanations into the filtering algorithm proposed by Fahimi et al. [7] by adapting Vilím's explanations [18]. We proposed two approaches to generating explanations: an approach using only the previously scheduled tasks to explain propagation and an approach using an even smaller subset of tasks combined with explanation lifting. An empirical evaluation of two of our approaches for generating explanations compared to a baseline with naïve explanations found that both approaches performed better in terms of conflicts, LBD, learned clause length and runtime. The most advanced approach of the two (last cluster) performed the best. We believe that using the last cluster approach to generating explanations with other propagators for the disjunctive constraint could be successful.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Constraint Programming, LCG, Disjunctive Constraint

1 Introduction

One of the most prevalent constraints used to model scheduling problems is the disjunctive constraint [16]. The disjunctive constraint enforces that two tasks using the same resource can not overlap. This constraint can for example be used to ensure that one lecture hall is not used by multiple lectures at the same time.

To solve such scheduling problems, constraint programming (CP) solutions are often considered. In CP, problems are modeled as a combination of variables and constraints before a solver is called to find a solution to the model. CP solvers often combine exhaustive search with domain-filtering (propagation) of variables to avoid exploring infeasible solutions. Propagators for the disjunctive constraint are used to ensure that the tasks within the constraint do not overlap. Propagation is performed by so-called filtering algorithms, which are based on certain propagation rules. One of these rules for the disjunctive constraint is detectable precedences. Detectable precedences is one of the simpler propagation rules for the disjunctive constraint, but it is sometimes able to find propagations which cannot be found by more complex rules such as edge-finding and not-first/not-last [16].

For detectable precedences, a filtering algorithm with a time complexity of $O(n \log n)$ is described by Vilím in 2004 [17]. In 2014, Fahimi et al. [7] published a linear time filtering algorithm. In a later paper, Fahimi et al. [6] recommended that future work should be done on adapting the algorithm to support explanations, which would then allow lazy clause generation (LCG) [9] to be used. This is a state-of-the-art CP solving technique, which has been the focus of much research in recent years. The usage of LCG necessitates the generation of explanations.

Explanations are reasons for certain propagations. A solver which generates explanations does not only need to prune the domains of variables, they also need to capture the reason for the modification of the domain. This reason is often in the form of a conjunction of atomic constraints involving the variables that are currently filtered [15].

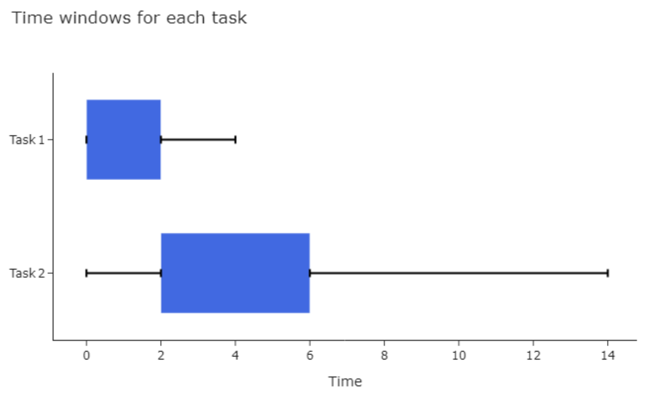

Figure 1 A graphical representation of the situation given in example 1

▶ **Example 1.** Suppose we have two tasks: $S_1 \in [0, 2]$ and $S_2 \in [0, 10]$ (task one can start at times 0-2 and task two can start at times 0-10). Task one takes two units of time to complete and task two four. The situation is sketched in figure 1. Because the earliest possible completion time of task two is greater than the latest possible starting time of task one, we can conclude that task one must precede task two. We can therefore propagate that $S_2 \geq 2$. An explanation for this propagation is the following: $S_1 \geq 0 \wedge S_1 \leq 2 \wedge S_2 \geq 0$.

Explanations for the disjunctive constraint have not been described extensively in the literature. While Vilím described a way to generate explanations for the disjunctive constraint in 2005 [18], we will see in the preliminaries (3.6) that Vilím's explanations can not be directly used in an LCG solver.

Considering that Fahimi et al. [6] recommended future work on evaluating their algorithm in an LCG setting and that explanations for the disjunctive constraint described in the literature cannot be directly used in an LCG solver, our research aims to fill this gap. We do so by describing how the linear time filtering algorithm proposed by Fahimi et al. [7] can be adapted to support explanations. We would also like to show the performance implications of using these explanations in a lazy clause generating solver.

We describe three approaches to generating explanations in the approach section. The first approach is a naïve baseline approach to generating explanations. The second approach is based on a simple observation we make in Fahimi's algorithm, namely that the previously scheduled tasks are sufficient to explain propagation. For the third approach we propose a method to combine Fahimi's algorithm with an adaptation of Vilím's way of generating explanations for detectable precedences. For this we show that a subset of the previously scheduled tasks satisfies the property of Ω' described by Vilím [18]. We then show how to record this set of tasks in Fahimi's algorithm [7]. Lastly, we describe how the explanation can be lifted in a way that is valid within an LCG solver by using an intermediate result from Vilím [18].

We evaluate the three approaches on a set of 50 jobshop instances. The results indicate that both non-baseline approaches outperform the naïve baseline approach in terms of number of conflicts, literal block distance, learned clause length and runtime. The metrics indicate that the third approach, called last cluster, performs best across all metrics.

To summarise our contributions, we described a way to generate explanations for detectable precedences in Fahimi's algorithm using an adaptation of Vilím's way of generating explanations. We also provide an empirical evaluation of two explanation strategies compared to a naïve baseline approach.

This paper is structured in the following way: in Section 2 we will discuss related work, followed by Section 3 describing the preliminary knowledge necessary for understanding the different approaches. Section 4 describes the three approaches that are used for benchmarking. Section 5 outlines our experiment, after which we give the conclusion and recommendations for future work in Section 6. Lastly, the responsible research section can be found in Appendix A.

2 Related work

Multiple propagation rules for the disjunctive constraint are described in the literature. Vilím's PhD thesis [16] describes filtering algorithms for overload checking, not-first/not-last, edge-finding and detectable precedences. The propagation rules edge-finding and not-first/not-last do not subsume detectable precedences, which is visualised in figure 2.6 of [16]. Because different rules can supplement each other, different propagators are often combined [16, 17].

Filtering algorithms using detectable precedences have been described in [17] and [7]. Vilím's paper [17] proposes an $O(n \log n)$ algorithm that incrementally constructs a balanced binary tree of all tasks that must be scheduled before the task we are considering. This Θ-tree also keeps track of the earliest completion time of all tasks scheduled on the tree, which is used as the new lower bound of the starting time of the following tasks. Fahimi et al. [7] proposes a similar algorithm, which uses a timeline data structure instead of a tree. This timeline data structure is based on a Union-Find data structure, which allows for $O(1)$ scheduling of tasks rather than Vilím's $O(\log n)$. This causes Fahimi's algorithm to run in linear time. In 2018 another publication by Fahimi et al. [6] suggested that future work incorporates explanations in the algorithms to evaluate its performance in a lazy clause generation solver. Neither Vilím's nor Fahimi's work describes how to explain the propagations made by the filtering algorithms.

However, previous work on explaining the disjunctive constraint has been done by Vilím in 2005 [18]. Here the author proposed to generate justifications based on conflict windows for not-first/not-last, edge-finding, and detectable precedences. Justifications are similar to the explanations we want to generate. These justifications can not be used directly in an LCG solver, because the justifications only explain that a propagation must occur, but it does not justify how much the earliest starting time of the propagated task is increased. In the preliminaries section we will see how Vilím's conflict windows can be adapted to explanations that can be used in our propagator.

There is some more recent work that was done on explaining propagators for the cumulative constraint by Yang et al. [19] and by Schutt et al. [15]. The cumulative constraint is a generalisation of the disjunctive constraint where limited overlap of tasks is allowed. Both papers provide explanations within a lazy clause generation context followed by an empirical evaluation in which the different explanation approaches are compared to each other. The results in both papers showed that explanations have a significant effect on the performance of the solver.

Our work aims to incorporate explanations into the detectable precedences algorithm proposed by Fahimi et al. [7]. We use an adaptation of Vilím's [18] justifications to explain propagations in a lazy clause generating context. To support these explanations, an addition to the timeline data structure used by Fahimi's algorithm is necessary.

3 Preliminaries

We provide background on Constraint Programming and Lazy Clause Generation in 3.1 and 3.2 respectively. To understand the different approaches for generating explanations we describe in the approach section, it is necessary to first understand the basic scheduling terminology. This is discussed in 3.3. Then we will briefly explain the detectable precedences rule in 3.4. For the third, most advanced, explanation approach we consider it necessary to be familiar with the algorithm described in 3.5 and Vilím's explanation approach in 3.6.

3.1 Constraint Programming

Constraint programming (CP) is an approach to solving combinatorial problems in which the problem is formulated as a combination of a set of variables $\mathcal{V}$, a set of domains $\mathcal{D}$ and a set of constraints $\mathcal{C}$. Constraints are rules that a valid assignment must adhere to. An assignment consists of a mapping of every variable $x \in \mathcal{V}$ to a value $d \in \mathcal{D}(x)$, where $\mathcal{D}(x)$ denotes the domain associated with x. We distinguish between constraint satisfaction problems (CSPs) in which any valid assignment needs to be found and constraint optimisation problems (COPs) in which the optimal valid assignment needs to be found. In COPs an objective function is added, which is used to assess how good an assignment is.

When solving CSPs and COPs, exhaustive search using backtracking is often combined with constraint propagation to make exhaustive search more tractable. In constraint propagation, the domains of variables involved in the same constraint are pruned if a constraint propagator can infer that assigning a variable a certain value from its domain would always result in a violation of the constraint. If a propagator prunes all remaining values from a domain, there does not exist a valid assignment to that variable and we encounter a conflict.

3.2 Lazy Clause Generation

Lazy clause generation (LCG) is a CP solver technique, which uses an embedded SAT engine to allow the usage of conflict analysis [9]. When a conflict is encountered, the reason for the conflict is processed and a *nogood* clause is learned. Instead of backtracking one level in regular backtracking, the solver can backjump to the decision level at which the nogood became true. In order to allow the usage of conflict analysis, propagators in an LCG solver need to record reasons for propagations and conflicts. These reasons are called *explanations*. Explanations are given as a conjunction of atomic constraints, which implies the propagation that is made. Explanations can often be made more general in order to learn more general nogoods. This is called explanation lifting. Feydy et al. [8] found that explanation lifting could improve solving time on certain benchmarks.

3.3 Scheduling terminology

We will consider $\mathcal{I}$ the set of all tasks that need to be scheduled. Every task $i \in \mathcal{I}$ is characterised by the following properties:

- est_i is the earliest starting time of i.
- lct_i is the latest completion time, or the deadline of i.
- p_i is the duration of task i.

In addition to these three defining properties, we also use the following notation:

- $lst_i = lct_i - p_i$ is the latest starting time of i.
- $ect_i = est_i + p_i$ is the earliest completion time of i.

The *est* notation can also be used on sets of tasks. Suppose we have a set of tasks Ω, then the following definitions hold for Ω:

- $est_\Omega = \min_{i \in \Omega} est_i$
- $lst_\Omega = \max_{i \in \Omega} lst_i$
- $p_\Omega = \sum_{i \in \Omega} p_i$

We will often use S to refer to the variable of the starting time of a task. Therefore, the following should hold: $S_i \in [est_i, lst_i]$

3.4 Detectable precedences

Given two distinct tasks, $i, j \in \mathcal{I}$ the precedence ($j \ll i$) is detectable if:

$$ect_i > lst_j$$

When we have detected a precedence $j \ll i$, we know that the earliest possible starting time for i must be at least as late as the earliest completion time of j. This gives us the following propagation rule:

$$est'_i = \max(ect_j, est_i)$$

The symmetric case also holds, but in this paper only the detectable precedences using the rule above are considered.

3.5 Timeline data structure

We will first explain the timeline data structure introduced by Fahimi et al. [7] and after that their proposed algorithm (algorithm 5) for detectable precedences will also be included.

The timeline data structure was introduced by Fahimi et al. [7] in 2014. They took inspiration from López-Ortiz et al. [14] who proposed a new algorithm for the alldifferent constraint, a specialisation of the disjunctive constraint.

The timeline data structure contains five fields:

1. t, an array containing all distinct earliest starting times of the tasks. A sufficiently large time is appended to allow all tasks to be scheduled on this timeline. The sufficiently large time used by Fahimi et al. is also the one we shall use:

$$\max_{i \in \mathcal{I}} lct_i + \sum_{i \in \mathcal{I}} p_i$$

2. c, an array keeping track of the capacities between different earliest starting times of tasks in t. At initialisation of the timeline, the following holds:

$$\forall k \in \{0, ..., |t| - 2\}\ c[k] = t[k+1] - t[k]$$

3. m, an array that maps the task with index i to its earliest starting time on t. So if the earliest starting time of task i is 3, $t[m[i]] = 3$ should hold.
4. s, a union find data structure that is used to keep track of the capacity between different starting times in t. s is initialized with $|t|$ elements and if a task is scheduled, which causes $c[a] = 0$, the sets of a and $a + 1$ are merged. This allows for $O(1)$ lookup of the first gap later than the earliest starting time of the task that needs to be scheduled.

5. e, a number keeping track of the highest index in c on which a task has been (partially) scheduled. This is used to calculate the earliest completion time of the timeline. e is initialised to -1.

An algorithm specifying how the timeline data structure is initialized is given in [7]. Here we will show the initialisation based on an example.

▶ **Example 2.** Consider three tasks defined by their tuples (est, p, lct). $\mathcal{I} = \{(1,5,8),(5,2,9),(5,3,15)\}$ Initialising the timeline gives us the following values for the fields:

- $t = [1, 5, 25]$
- $c = [4, 20]$
- $m = [0, 1, 1]$
- $s = \{\{0\}, \{1\}, \{2\}\}$
- $e = -1$

We use the following notation to display the current timeline with its capacity: $\{1\} \xrightarrow{4} \{5\} \xrightarrow{20} \{25\}$

3.5.1 Task scheduling

Scheduling of tasks is performed through the following algorithm. This algorithm is algorithm 3 in [7].

Algorithm 1 ScheduleTask(i)

```
$\rho \leftarrow p_i$
$k \leftarrow s.\textsc{FindGreatest}(m[i])$
while $\rho > 0$ do
    $\Delta \leftarrow \min(c[k], \rho)$
    $\rho \leftarrow \rho - \Delta$
    $c[k] \leftarrow c[k] - \Delta$
    if $c[k] = 0$ then
        $s.\textsc{Union}(k, k+1)$
        $k \leftarrow s.\textsc{FindGreatest}(k)$
    end if
end while
$e \leftarrow \max(e, k)$
```

▶ **Example 3.** If we consider the timeline from the previous example, we get the following timelines when scheduling the tasks:

- $\{1\} \xrightarrow{4} \{5\} \xrightarrow{20} \{25\}$, before any tasks are scheduled.
- $\{1, 5\} \xrightarrow{19} \{25\}$, after scheduling the first task
- $\{1, 5\} \xrightarrow{17} \{25\}$, after scheduling the second task
- $\{1, 5\} \xrightarrow{14} \{25\}$, after scheduling the third task

The earliest completion time of the current timeline can be computed by evaluating the following expression:

$$ECT_\Theta = t[e+1] - c[e]$$

After the third task has been added, $ECT_\Theta = 11$

3.5.2 Detectable precedences filtering algorithm

All timeline preliminary knowledge builds up to the following filtering algorithm, which uses the detectable precedences rule. This algorithm is algorithm 5 in [7]. It works by considering tasks sorted on ect for propagation in the outer loop and the tasks in the inner while loop sorted on lst are scheduled on the timeline if they precede task i based on the detectable precedences rule. Since tasks with a compulsory part $lst_k < ect_k$ could be scheduled on the timeline before being considered for filtering, scheduling needs to be postponed until the blocking task k is visited in the outer loop.

$\mathcal{I}_{lst}$ denotes the set of tasks $\mathcal{I}$ sorted by latest starting time in ascending order.

Algorithm 2 DetectablePrecedences($\mathcal{I}$)

```
InitializeTimeline($\mathcal{I}$)
$j \leftarrow 0$
$k \leftarrow \mathcal{I}_{lst}[j]$
$postponed\_tasks \leftarrow \varnothing$
$blocking\_task \leftarrow null$
for $i \in \mathcal{I}_{ect}$ do
    while $j < |\mathcal{I}| \wedge lst_k < ect_i$ do
        if $lst_k \geq ect_k$ then ScheduleTask($k$)
        else
            if $blocking\_task \neq null$ then return Inconsistent
            end if
            $blocking\_task \leftarrow k$
        end if
        $j \leftarrow j + 1$
        $k \leftarrow \mathcal{I}_{lst}[j]$
    end while
    if $blocking\_task = null$ then
        $est'_i \leftarrow \max(est_i,$ EarliestCompletionTime()$)$
    else
        if $blocking\_task = i$ then
            $est'_i \leftarrow \max(est_i,$ EarliestCompletionTime()$)$
            ScheduleTask($blocking\_task$)
            for $z \in postponed\_tasks$ do
                $est'_z \leftarrow \max(est_z,$ EarliestCompletionTime()$)$
            end for
            $blocking\_task = null$
            $postponed\_tasks \leftarrow \varnothing$
        else
            $postponed\_tasks \leftarrow postponed\_tasks \cup \{i\}$
        end if
    end if
end for
for $i \in \mathcal{I}$ do
    $est_i \leftarrow est'_i$
end for
```

To illustrate this algorithm we give the following example:

▶ **Example 4.** Suppose we have four tasks defined by their triple (est, p, lct):

1. Task 1: (0, 2, 6) with $lst = 4$ and $ect = 2$
2. Task 2: (3, 5, 10) with $lst = 5$ and $ect = 8$
3. Task 3: (7, 5, 15) with $lst = 10$ and $ect = 12$
4. Task 4: (4, 2, 20) with $lst = 18$ and $ect = 6$

The situation is also sketched in figure 2 We give a full algorithm walkthrough in Appendix B.

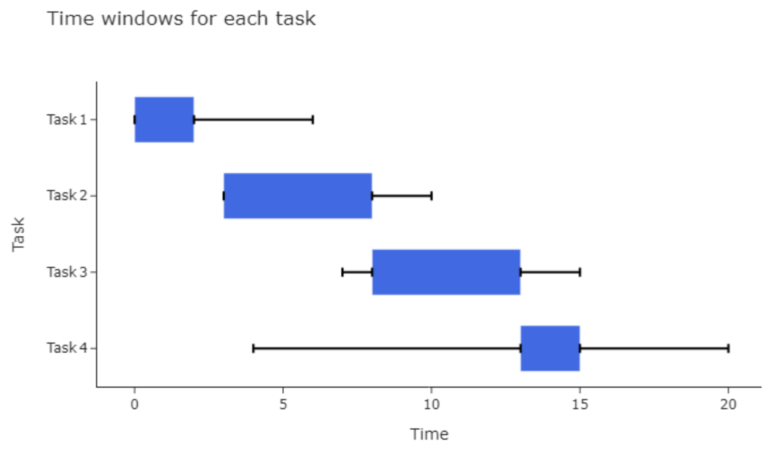

Figure 2 The situation described in example 4. Note that the order of tasks 3 and 4 could be switched as this does not produce an infeasible schedule.

After applying the algorithm, it can be found that the adjusted earliest starting times of the tasks are the following:

1. $est_1 = 0$
2. $est_2 = 3$
3. $est_3 = 8$
4. $est_4 = 8$

3.6 Vilím's explanations for detectable precedences

Vilim's 2005 paper [18] describes ways to generate explanations for propagators for the disjunctive constraint. This paper uses conflict windows, which differ from our typical way of using conjunctions of atomic constraints for explanations.

Conflict windows are given with the following notation: $\langle \underline{est_i}, \underline{lct_i} \rangle$. They can be thought of as relaxations of the bounds of a task, which are set as wide as possible without making the fact that a propagation occurs invalid.

We will now explain how Vilím proposed to set the conflict windows. Suppose Θ is a data structure that holds the set of all previously scheduled tasks based on the detectable precedences rule. We need to find $\Omega' \subseteq \Theta$ for which it holds that $ECT_\Theta = est_{\Omega'} + p_{\Omega'}$ where ECT_Θ is the earliest completion time calculated using data structure Θ.

Assuming that we have found Ω', according to Vilím [18] we can generate the following conflict windows for the detectable precedences $\Omega' \ll i$: Let $\Delta = est_i + p_i - \max_{j \in \Omega'}(lct_j - p_j) - 1$

- i: $\langle est_i - \lceil \frac{\Delta}{2} \rceil, \infty \rangle$
- $\forall j \in \Omega'$: $\langle \underline{est_i} - p_{\Omega'}, est_i + p_i + p_j - \lceil \frac{\Delta}{2} \rceil - 1 \rangle$

▶ **Example 5.** Consider our previous example, with $S_1 \in [0, 2]$, $p_1 = 2$, $S_2 \in [0, 10]$ and $p_2 = 4$. We can then generate the following conflict windows: $\langle \underline{est_2}, \underline{lct_2} \rangle = \langle -1, \infty \rangle$ and $\langle \underline{est_1}, \underline{lct_1} \rangle = \langle -3, 4 \rangle$

In order to use the conflict window as bounds in our explanations, we should convert $\underline{lct}$ to $\underline{lst}$, which gives us $S_1 \geq -3 \wedge S_1 \leq 2 \wedge S_2 \geq -1$. Please note that this is not a valid explanation as $S_1 \geq -3 \wedge S_1 \leq 2 \wedge S_2 \geq -1 \not\Rightarrow S_2 \geq 2$. The reason for this is that we cannot conclude that $S_2 \geq 2$ if we only know that $S_1 \geq -3$, because scheduling S_1 at time -3 does not 'push' task 2 far enough to guarantee $S_2 \geq 2$.

We can not use Vilím's weakened conflict windows directly, but an intermediate result described in Vilím's derivation of conflict windows could be used. An intermediate result in Vilím's derivation namely suggests $\forall j \in \Omega'$ $\underline{est_j} = est_{\Omega'}$. Since we propagate that $S_i \geq ECT_\Theta$ and because $ECT_\Theta = est_{\Omega'} + p_{\Omega'}$, we can lift the earliest starting time for all tasks in Ω' to $est_{\Omega'}$ without invalidating the propagation.

4 Approach

As the aim of this paper is to generate explanations for the detectable precedences filtering algorithm by Fahimi et al. [7], we need to take into account two important factors. The first factor we consider is time complexity, because it might not be worthwhile to generate explanations if the overhead caused by recording explanations is too great. The second factor is the generality of the explanations, as more general explanations have been shown to outperform less general explanations [8].

In order to measure the influence of different explanations on the number of conflicts and other relevant metrics, we will consider three different approaches to generating explanations. The baseline approach is completely naïve with constant time overhead for generating explanations. The intermediate approach (previously scheduled) generates explanations that are more general than baseline, while also only taking constant time to record. The last cluster approach describes the most lifted explanations, which come at the cost of linear time overhead for each propagation.

4.1 Naïve explanations

Naïve explanations are often used as a baseline to compare with more sophisticated ways of generating explanations. In naïve explanations, the reason for every propagation is the same, namely:

$$\bigwedge_{i \in \mathcal{I}} S_i \geq est_i \wedge S_i \leq lst_i$$

Since this type of explanation is likely to contain atomic constraints which are not necessary to explain the propagation and because the bounds of the atomic constraints might be lifted to include more values, this method does not produce general explanations.

▶ **Example 6.** Recall example 4, where we propagated that $S_3 \geq 8$ and $S_4 \geq 8$. The naïve explanation for both of these propagations is $S_1 \geq 0 \wedge S_1 \leq 4 \wedge S_2 \geq 3 \wedge S_2 \leq 5 \wedge S_3 \geq 7 \wedge S_3 \leq 10 \wedge S_4 \geq 4 \wedge S_4 \leq 18$

As we only need to compute the naïve explanation once before we can use it to explain all propagations made by the filtering algorithm, the overhead is constant.

4.2 Previously scheduled tasks

By taking a closer look at algorithm 2, we can find that only the tasks that were previously scheduled on the timeline have any influence on the lower bound adjustment of a starting time's domain. We can also omit the upper bound of the task we are pruning, because the detectable precedences rule $ect_i > lst_j$ only depends on the earliest completion time of the task we are pruning. If we consider $\Theta \subseteq \mathcal{I}$ to be the tasks that are currently scheduled on the timeline and i is the task we are pruning, we can generate the following explanations:

$$S_i \geq est_i \bigwedge_{j \in \Theta} S_j \geq est_j \wedge S_j \leq lst_j$$

▶ **Example 7.** When we once again consider example 4, we can generate the following explanations:

- For propagation $S_3 \geq 8$: $S_3 \geq 7 \wedge S_1 \geq 0 \wedge S_1 \leq 4 \wedge S_2 \geq 3 \wedge S_2 \leq 5$
- For propagation $S_4 \geq 8$: $S_4 \geq 4 \wedge S_1 \geq 0 \wedge S_1 \leq 4 \wedge S_2 \geq 3 \wedge S_2 \leq 5$

Because we can incrementally build up an explanation whenever a new task is scheduled, the overhead caused by recording the previously scheduled explanations is constant.

4.3 Last cluster approach

In the preliminaries it is described that a set of tasks $\Omega' \subseteq \Theta$ such that $ECT_\Theta = est_{\Omega'} + p_{\Omega'}$ is sufficient to explain a propagation. We will first explain how we can obtain Ω' from the timeline data structure. Then we will describe how we lift the bounds of the predicates in Ω' to generate explanations that are more general.

4.3.1 Recording Ω'

Let us define the last cluster of the timeline data structure to be the set in the timeline's union find which contains the highest starting time for which a task is currently scheduled. The timeline already keeps track of this cluster with field e. We will illustrate that the tasks in the last cluster are the only ones that are responsible for determining ECT_Θ

▶ **Example 8.** Suppose we have the following timeline: $\{0\} \xrightarrow{3} \{5\} \xrightarrow{7} \{15\}$. We can infer that at least one task is scheduled at $est = 0$ and that at least one task is scheduled at $est = 5$. $e = 1$ in this case. As $ECT_\Theta = t[e+1] - c[e]$ in the timeline data structure, $ECT_\Theta = 8$. We can see that ECT_Θ is independent of any tasks not scheduled in the last cluster.

Now that we know that tasks in the last cluster are the only ones that are responsible for ECT_Θ, we will show that the set of tasks in the last cluster satisfies the property of Ω'. Recall that $\Omega' \subseteq \Theta$ such that $ECT_\Theta = est_{\Omega'} + p_{\Omega'}$. Since any number of tasks in a cluster is scheduled contiguously (otherwise the tasks would not be in the same cluster), the earliest completion time of the last cluster A must be $est_A + p_A$. Combining this with our previous observation that the last cluster is solely responsible for ECT_Θ, we can conclude that the tasks of the last cluster satisfy Ω'.

▶ **Observation 9.** *The last cluster set A satisfies the property of $\Omega' \subseteq \Theta$ where $ECT_\Theta = est_{\Omega'} + p_{\Omega'}$*

Having found that the last cluster satisfies Ω', we now need to solve the problem of recording which tasks are scheduled in the last cluster. To address this, we propose to add a field u to the timeline. u is an array consisting of $|t| - 1$ lists, where every list at index i in u

keeps track of which tasks are scheduled at the starting time corresponding to $t[i]$. Please note that as it is impossible to schedule a task at the latest time in t, we do not need to keep track of any tasks scheduled at that time. That is why the length of u is $|t| - 1$.

The main idea behind u is that we want to 'mirror' the state of s, the union find. We achieve this by applying the following rules:

- In algorithm 1 we add the task we scheduled to $u[k]$ after the while loop.
- Whenever k and $k + 1$ are unioned in algorithm 1, all elements in $u[k]$ are added to $u[s.find(k + 1)]$
- Ω' can be retrieved at $u[e]$

An adapted version of Algorithm 1 with support for recording Ω' can be found in the appendix under the name Algorithm 3.

4.3.2 Generating explanations

Now that we are able to retrieve Ω' from the timeline, we are ready to define how we can generate explanations for our propagations. In the preliminaries we have found that for any task in Ω' we can not lift the earliest starting time further than $est_{\Omega'}$. This was an intermediate result in Vilím's [18] derivation for the conflict windows. For the latest starting time we can choose to set it to $lst_{\Omega'}$. Intuitively, this allows the tasks in Ω' to be scheduled in any order. If we lift the latest starting time of tasks in Ω' to $lst_{\Omega'}$, the furthest we can lift the earliest starting time of task i without violating the detectable precedences rule is $lst_{\Omega'} - p_i + 1$.

This leaves us with the following explanation:

$$S_i \geq lst_{\Omega'} - p_i + 1 \bigwedge_{j \in \Omega'} S_j \geq est_{\Omega'} \land S_j \leq lst_{\Omega'}$$

▶ **Example 10.** We once again consider example 4, we can generate the following explanations:

- For propagation $S_3 \geq 8$: $S_2 \geq 3 \land S_2 \leq 5 \land S_3 \geq 1$
- For propagation $S_4 \geq 8$: $S_2 \geq 3 \land S_2 \leq 5 \land S_4 \geq 4$

Because the bounds $est_{\Omega'}$ and $lst_{\Omega'}$ can be different for every propagation, the overhead caused by recording last cluster explanations is linear. This is because we cannot incrementally build up an explanation like in the previously scheduled approach.

5 Experimental results

Since it is our goal to evaluate the influence of the different explanation approaches on the number of conflicts encountered while solving, we perform an experiment where we compare the performance of the previously scheduled approach 4.2 and the last cluster approach 4.3 to the baseline naïve approach 4.1. We found that on average the number of conflicts are 43% and 4% of the baseline for the previously scheduled and the last cluster approach respectively.

The section will continue by describing the experimental setup that was used, after which the metrics that are collected are covered. Lastly, we present the results of the experiment.

5.1 Experimental setup

A propagator using detectable precedences as it is described in algorithm 2 was implemented in the Pumpkin solver [5]. Three strategies for giving explanations were also implemented:

The naïve approach given in subsection 4.1, the previously scheduled approach given in subsection 4.2 and the last cluster approach given in 4.3. These three strategies were benchmarked on 50 jobshop instances given on the Minizinc benchmarks Github page[1]. The specific instances that were run are the following: abz5-abz7 [1], ft06 and ft10 [12], la01-la35 [10] and orb01-orb10 [2]. The reason these instances were chosen is because we wanted all approaches to be able to make a reasonable amount of progress before the timeout, as a benchmark instance would not be of much use if no new solutions are found after the first couple of seconds.

The benchmarks were run through a Minizinc [13] front-end with a jobshop model file, that is adapted to use the disjunctive constraint. The branching strategy that was used is free search with a search annotation which selects the starting time variable with the smallest value and assigns the smallest value in the domain. As we use free search, the solver switches to VSIDS after the first solution is found. VSIDS is a branching technique that is biased to select unassigned variables that occur often in recent conflicts [11]. The reason we chose this search strategy is because it outperformed any alternatives. The experimental setup including benchmarks is published in a Github repository [2]. The code of the Pumpkin solver which was used is also published in a Github repository [3].

The hardware used to run the benchmarks is an HP ZBook Power G9 with a 12th Gen Intel(R) Core(TM) i7-12700H processor.

5.2 Metrics

For every benchmark instance the following four metrics are collected:

1. **Number of conflicts**: the total number of conflicts that were encountered by the solver.
2. **Average Literal Block Distance (LBD)**: LBD is a metric that was introduced for SAT solving in [3] and it is defined as the number of unique decision levels at which the predicates in a learned nogood have become true. A lower LBD means that less decisions were responsible for learning a certain nogood, which means that it is more likely to be reused than a nogood with a higher LBD.
3. **Average learned clause length (LCL)**: LCL refers to the number of predicates that are present in the learned clause. The average LCL compared with the LBD provides insight into the average number of predicates in a learned clause that originate from the same decision point.
4. **Runtime**: the amount of time the solver took to reach the objective value at which approaches are compared.

In the experiment, the metrics for all three approaches are compared at the best objective which was reached by all approaches.

5.3 Results

The results of the experiment can be found in table 1. What stands out immediately is the average number of conflicts. For the previously scheduled approach it is half the number of conflicts of the baseline. The number of conflicts encountered by the last cluster approach is one order of magnitude smaller than that of the previously scheduled approach. Based

[1] https://github.com/MiniZinc/minizinc-benchmarks/tree/master/jobshop
[2] https://github.com/MatvV04/DetectablePrecedencesExperiment/tree/main
[3] https://github.com/MatvV04/Pumpkin

	Naïve (baseline)	Previously scheduled	Last cluster
Average number of conflicts	45237.14	19478.14	1801.24
Average conflicts ratio with baseline (#conflicts approach / #conflicts baseline)	1	0.926	0.41
Average literal block distance	21.31	19.09	8.62
Average learned clause length	38.67	31.73	14.79
Average runtime ratio with baseline (runtime approach / runtime baseline)	1	0.822	0.773

Table 1 Aggregate results for the 50 benchmark instances for three different approaches. Average LBD and Average LCL are averages over averages. All statistics are compared at the best common found objective after 20 minutes.

on the average number of conflicts, we would expect that the average conflicts ratio with baseline would be smaller for both values than they are in reality (approximately 0.50 and 0.05 respectively based on the average number of conflicts). To explain why the average conflicts ratio with the baseline is relatively high, we plotted the number of conflicts of the two approaches against the number of conflicts of the baseline approach in the left part of figure 3. We can see that when the number of conflicts for baseline is quite low, the two other approaches generally do not differ much. In the extreme cases where the number of conflicts is high for the naïve approach, it tends to be much lower for the last cluster approach.

When we consider the right plot in figure 3, we can see that the last cluster approach generally has a lower runtime than the naïve baseline and that the the previously scheduled approach is on average only slightly better. If we focus on the top right of the plot, we can see that there are seven last cluster instances against the ceiling of 1200 seconds. The instances where this happens are abz7, la27, la29, la30, la32, la34, la35. These instances are among the largest in our benchmark suite and the best common lower bound that was found out of these seven instances was 193% of the optimal objective. We believe that the overhead of the last cluster approach might not outweigh the benefit of the more general explanations if the problem is not yet sufficiently constrained by the best found objective value.

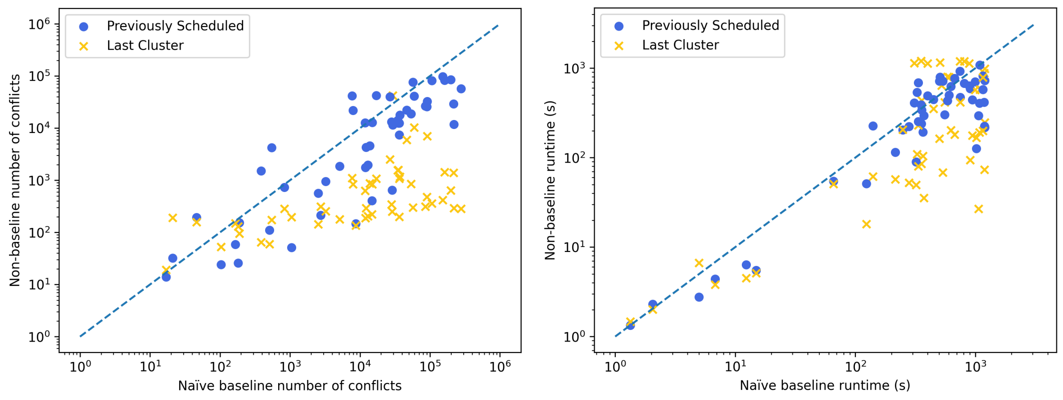

Figure 3 The number of conflicts (left) and runtime (right) of the two approaches on the y-axis plotted against the number of conflicts / runtime of the baseline naïve approach on the x-axis for the 50 jobshop instances. If a point is below the line, the metric of the non-baseline approach is lower than baseline.

When comparing the average LBD and LCL for the three approaches, it can be seen that it is lower than baseline for the previously scheduled approach and much lower for the last cluster approach. This indicates that the learned nogoods are on average caused by less decisions and containing less predicates, which make the nogoods more reusable.

The average runtime to reach the common best objective is also lower than that of the baseline approach for both the previously scheduled approach and the last cluster approach. On average the previously scheduled approach is 17.8% faster and the last cluster approach 22.7%.

Next, we perform a pairwise Student t-test on the number of conflicts between the baseline and the other two approaches. The null hypothesis is that the number of conflicts does not differ from the naïve baseline approach. The alternative hypothesis is that the number of conflicts is lower than the baseline. Before performing the test, the significance level was set to 0.05. The results can be found in table 2. According to our set significance level, we reject the null hypothesis for both approaches. This means that there is significant statistical evidence that the number of conflicts for both approaches is lower than that of the naïve baseline.

	Previously scheduled	Last cluster
p-value	0.00628	6.657E-6

Table 2 The p-values computed by a pairwise Student t-test for the number of conflicts compared to baseline

A comparison between the last cluster approach and a decomposition is given in Appendix D. The most important result of this comparison is that the decomposition is on average 18 times faster than the last cluster approach, but the decomposition appears to have difficulty proving optimality compared to the last cluster approach.

6 Conclusion

We proposed two approaches to generating explanations for the filtering algorithm proposed by Fahimi et al. [7]. The first approach considers only the subset of tasks that have previously been scheduled on the timeline. The second approach only considers the last cluster of the previously scheduled subset. Explanation lifting was also described for this approach. An experiment comparing the previously scheduled approach and the last cluster approach to the baseline naïve approach for generating explanations found that the number of conflicts for both approaches was lower than the baseline. This finding was substantiated by a Student t-test with a significance level of 5%. The last cluster approach outperformed the previously scheduled approach by 10 times in terms of average number of conflicts. The other metrics, LBD, LCL and runtime, were also lower for the previously scheduled approach and more noticeably lower for the last cluster approach. The last cluster did not outperform the other approaches in terms of runtime on seven large instances, which we believe is due to the overhead of recording the explanations not outweighing the more general explanations when the problem is not yet sufficiently constrained by the objective value. Further experimentation could be done with a higher timeout to observe how the three approaches perform on larger instances when the problem is solved closer to optimality.

This research only considered explanations for detectable precedences in isolation, but as different propagation rules are often combined, future work should focus on the performance of combining different disjunctive propagation rules in a lazy clause generation solver.

References

1 Joseph Adams, Egon Balas, and Daniel Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401, 1988. URL: `http://www.jstor.org/stable/2632051`.

2 D Appelgate and W Cook. A computational study of jobshop scheduling. *ORSA Journal on Computing*, 3, 1991.

3 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, page 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

4 Nicolas Beldiceanu, Pierre Flener, Jean-Noël Monette, Justin Pearson, and Helmut Simonis. Toward sustainable development in constraint programming. *Constraints*, 19(2):139–149, April 2014. `doi:10.1007/s10601-013-9152-4`.

5 Emir Demirović, Maarten Flippo, Imko Marijnissen, Konstantin Sidorov, and Smits Jeff. Pumpkin: A lazy clause generation constraint solver in rust, 2024. URL: `https://github.com/ConSol-Lab/Pumpkin`.

6 Hamed Fahimi, Yanick Ouellet, and Claude-Guy Quimper. Linear-time filtering algorithms for the disjunctive constraint and a quadratic filtering algorithm for the cumulative not-first not-last. *Constraints*, 23(3):272 – 293, 2018. Cited by: 10. URL: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-85048598742&doi=10.1007%2fs10601-018-9282-9&partnerID=40&md5=3670ec8a382d13e04e0e34b8189b35f2`, `doi:10.1007/s10601-018-9282-9`.

7 Hamed Fahimi and Claude-Guy Quimper. Linear-time filtering algorithms for the disjunctive constraint. volume 4, page 2637 – 2643, 2014. Cited by: 6. URL: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-84908192155&partnerID=40&md5=da5a1644a8b3303a379a766e635dce14`.

8 Thibaut Feydy, Andreas Schutt, and Peter Stuckey. Semantic learning for lazy clause generation. In *TRICS workshop, held alongside CP*. Citeseer, 2013.

9 Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, pages 352–366, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

10 Stephen Lawrence. Resouce constrained project scheduling: An experimental investigation of heuristic scheduling techniques (supplement). *Graduate School of Industrial Administration, Carnegie-Mellon University*, 1984.

11 M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535, 2001. `doi:10.1145/378239.379017`.

12 John F Muth, Gerald Luther Thompson, and Peter R Winters. Industrial scheduling. *(No Title)*, 1963.

13 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

14 Alejandro Opez-ortiz, Claude-guy Quimper, John Tromp, and Peter van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. *IJCAI International Joint Conference on Artificial Intelligence*, 05 2003.

15 Andreas Schutt, Thibaut Feydy, Peter Stuckey, and Mark Wallace. Explaining the cumulative propagator. *Constraints*, 16:250–282, 07 2011. `doi:10.1007/s10601-010-9103-2`.

16 Petr Vilím. Global constraints in scheduling. 2007.

17 Petr Vilím. O(n log n) filtering algorithms for unary resource constraint. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3011:335 – 347, 2004. Cited by: 32. URL: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-35048841193&doi=10.`

1007%2f978-3-540-24664-0_23&partnerID=40&md5=5245d259a905346cd6608260b668943a, doi:10.1007/978-3-540-24664-0_23.

18 Petr Vilím. Computing explanations for the unary resource constraint. volume 3524, page 396 – 409, 2005. Cited by: 6. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-26444453950&doi=10.1007%2f11493853_29&partnerID=40&md5=cf493e7730ec0a0e274a1f3ed0f0ff48, doi:10.1007/11493853_29.

19 Moli Yang, Andreas Schutt, and Peter J. Stuckey. Time table edge finding with energy variables. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11494 LNCS:633 – 642, 2019. Cited by: 2. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85066859197&doi=10.1007%2f978-3-030-19212-9_42&partnerID=40&md5=183b549bebb587cd63fd7161f4bda8e4, doi:10.1007/978-3-030-19212-9_42.

A Responsible research

This section will discuss the ethical considerations and the measures taken to ensure the reproducibility of the results in our research. We also reflect on the integrity of our research.

The ethical implications of doing research into more efficient solving of constraint programming were considered. More efficient CP solving could enable problems in operations research to be solved more efficiently or closer to optimality. This can have both positive and negative ethical implications. An example of the positive implications would be more efficient usage of resources, while a negative implication could be that automation due to progress in operations research could cause workers to be made redundant. We believe that within this research there have been no opportunities to mitigate the negative ethical implications associated with more efficient solving of constraint programming other than to be transparent about the considerations.

Regarding ethics in the sustainable development of CP solvers, 15 challenges were published in [4]. The first challenge is the following:

> How can CP solver developers simplify the migration of the knowledge embedded in their solver to next-generation solvers?

One of the proposed mitigations is the usage of open-source solvers. As all experimental approaches were implemented in the Pumpkin solver [5], which is open source, we adhered as best as we could to these best practices.

The first measure we took to ensure the reproducibility of the results was choosing to only use benchmarks described in the literature. Other research such as [7] also makes use of randomly generated instances, which makes it difficult to reproduce the results, especially when these instances are not published in a repository.

To further simplify reproduction of the results, we have set up a Github repository for the code used to run the experiments. We also published a repository containing the Minizinc model file and the data files representing the benchmark instances along with the script used to run the benchmarks. Links to these repositories can be found in 5.1. These repositories also contain the raw data and scripts that are used to process the data and to produce the aggregates and plots. This is to ensure that it can be detected if mistakes were made in the data analysis. The README file in this repository also contains a description of the files present in the repository to simplify working with the data or reproducing results using the scripts.

Regarding research integrity, we included all results obtained through experimentation and did not cherry pick results to calculate the aggregates. We also reflect on results which at first seem to contradict each other by providing possible explanations for the results. Generative artificial intelligence (Github Copilot)[4] was used in this research only to aid in programming. Github Copilot was only used in the form of autocomplete and any code produced by it was checked manually.

B Complete example of algorithm 2

We give the complete working of example 4:

Suppose we have four tasks defined by their triple (est, p, lct):

- Task 1: (0, 2, 6) with $lst = 4$ and $ect = 2$
- Task 2: (3, 5, 10) with $lst = 5$ and $ect = 8$
- Task 3: (7, 5, 15) with $lst = 10$ and $ect = 12$
- Task 4: (4, 2, 20) with $lst = 18$ and $ect = 6$

The situation is also sketched in figure 4.

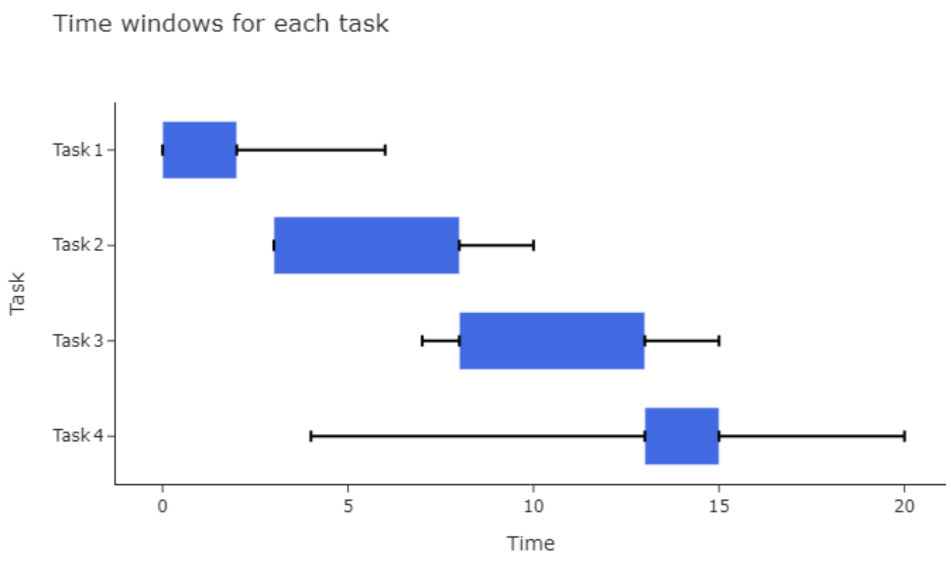

Figure 4 The situation described in the example

We give the following algorithm walkthrough:

Initialize timeline:

$\{0\} \xrightarrow{3} \{3\} \xrightarrow{1} \{4\} \xrightarrow{3} \{7\} \xrightarrow{27} \{34\}$

$k = 1$

Outer loop $i = 1$

$est'_1 = 0$

Outer loop $i = 4$

Inner loop:

Schedule task 1: $\{0\} \xrightarrow{1} \{3\} \xrightarrow{1} \{4\} \xrightarrow{3} \{7\} \xrightarrow{27} \{34\}$

[4] https://github.com/features/copilot

$k = 2$
Next iteration
$blocking\_task = 2$
$k = 3$
End inner loop
$postponed\_tasks = \{4\}$
Outer loop $i = 2$
$est'_2 = 3$
Schedule task 2: $\{0\} \xrightarrow{1} \{3, 4, 7\} \xrightarrow{26} \{34\}$
Inner for loop $z = 4$:
$est'_4 = 8$
Outer loop $i = 3$
Inner loop:
$blocking\_task = 3$
$k = 4$
End inner loop
$est'_3 = 8$
So we can propagate that $est_3 = 8$ and $est_4 = 8$.

C Adaptation of algorithm 1

Here we give the adapted algorithm for ScheduleTask in which we add functionality for recording Ω'

Algorithm 3 ScheduleTask(i) (Adapted)

```
$\rho \leftarrow p_i$
$k \leftarrow s.\text{FindGreatest}(m[i])$
while $\rho > 0$ do
    $\Delta \leftarrow \min(c[k], \rho)$
    $\rho \leftarrow \rho - \Delta$
    $c[k] \leftarrow c[k] - \Delta$
    if $c[k] = 0$ then
        $s.\text{Union}(k, k+1)$
        $k\_old \leftarrow k$
        $k \leftarrow s.\text{FindGreatest}(k)$
        $u[k].\text{Extend}(u[k\_old])$
    end if
end while
$u[k] \leftarrow i$
$e \leftarrow \max(e, k)$
```

D Comparison between last cluster and decomposition

To supplement our comparison with the naïve baseline, we also decided to compare the performance of the last cluster approach with a decomposition of the disjunctive constraint. The benchmark for the disjunctive constraint was run using the same search strategy as the last cluster approach, which was described in Subsection 5.1. The results presented in table 3 are based on 44 instances, as there was no common lower bound for six instances. In these instances (la29 and la31-la35) the decomposition vastly outperformed the last cluster approach in terms of reached objective value.

	Last cluster	Decomposition
Average number of conflicts	9635	18969
Average LBD	8.77	7.20
Average LCL	16.69	13.31
Average runtime (s)	362	20
Number of instances solved to optimality	37	28

Table 3 The last cluster approach compared to a decomposition on 44 instances with a common objective values, all instances were given a timeout of 20 minutes.

The most remarkable result in table 3 is the average runtime. The decomposition is on average 18 times faster than the last cluster approach. This could indicate that the propagation power of detectable precedences without any additional propagation rules is not great enough.

The second interesting result from table 3 is the number of instances solved to optimality. The decomposition failed to prove optimality for la06-la15, while the last cluster succeeded to do so. What is especially interesting is that the decomposition managed to find the optimal solution in all instances la06-la15 within ten seconds, but it failed to prove optimality within 1200 seconds. This is an indication that using a disjunctive propagator with general explanations might perform better than decomposition at proving optimality.