More general explanations for the Not-First/Not-Last propagators

Yousef El Bakri[1]
Supervisor(s): 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: Yousef El Bakri
Final project course: CSE3000 Research Project
Thesis committee: Emir Demirović, Imko Marijnissen, Stephanie Wehner

More general explanations for the Not-First/Not-Last propagator

Yousef El Bakri
Technical University Delft, Netherlands

Abstract

Currently explanations for the Not-First/Not-Last propagators for the disjunctive constraint have not been explored thoroughly, and have room for improvement. In this paper, we look into attempting to give more general explanations by looking through the powerset of tasks and finding a smaller set which still propagates. We have implemented naive, the state-of-the-art and our subset-finding explanations and compared them all. Experimentally, around 50% of the results show a lower amount of conflicts and average literal bound distance, however a majority have a higher runtime. In addition, the more subsets we consider the bigger the runtime, however it not necessarily decrease in amount of conflicts and average literal bound distance.

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

Keywords and phrases Constraint Programming, Job Scheduling, Disjunctive, Explanations, Not-First/Not-Last

1 Introduction

Constraint Programming is a method that is used to solve NP-Hard problems by modelling them. With the use of the variable along with the constraints, it can effectively find solutions without telling it *how* to find them. This method is used in many places, for example in train management for better fuel consumption [6]. Given their relevant use, looking into ways to optimize the algorithm would lead to faster solutions, which is of great importance.

In constraint programming, problems are modelled via variables and constraints. Given that a variable represents something in the problem, we can constraint it based off of the specifications of said problem. There are many constraints that can affect variables, however the two most important ones in task scheduling are the *disjunctive* and *cumulative* constraints. The disjunctive constraints allows for only one task to be executed at a time without disruption, while the cumulative constraint allows for multiple tasks at once without disruption.

Propagators are a part of Constraint Programming solvers which help speed up the solving process. By looking through the possible values of variables from their given domains, they can filter the ones that can not satisfy the constraints they are in. For each constraint there are many propagators that help prune the domains of variables, for example in disjunctive there is Edge-Finding and Detectable Precedences [16]. In our case, we will primarily focus on the Not-First/Not-Last propagator.

When filtering the domain or encountering an infeasibility, an explanation is generated in the form of a SAT problem to justify it [5]. Afterwards they get converted into learnt-clauses, which help the solver converge to a solution faster. How good an explanation is depends on primarily two things, the amount of variables it involves and amount of elements it potentially covers in the domain [12].

There has been tremendous progress since the first implemenetion of the Not-First/Not-Last propagators and their explanations. The main method is by using Θ trees to efficiently propagate within $\mathcal{O}(n \log n)$ [15]. Afterwards, they are used to generate explanations to justify the propagations [1], either by including the current domains of all tasks (naive), or further lifting them until there is still a conflict (Petr Vilím's explanations [1]). However

the naive and lifted explanations currently use all tasks which may cause a propagation for the explanation, rather than attempting to use the minimal set of variables required for propagation. This can lead to less effective learnt clauses being generated [12], decreasing overall performance.

In this paper, we will explore generating more general conflict windows for tasks by looking at smaller sets of tasks that affect propagation, with the goal of reducing the amount of conflicts and average literal bound distance [12]. In order to achieve this, we will look through all of the subsets of the set of tasks that could affect propagation, and picking the one that causes the biggest change. However since this is $\mathcal{O}(2^n)$ in time and space complexity, we will only be executing this when the set is smaller than a constant number. As such, a range of constants from two to five will be compared with a naive set of explanations and Petr Vilím's method.

Experimentally, the explanations introduced in this paper have shown promising results. While few instances were faster than the state-of-the-art implementation, we found that 50% of test instance encountered less conflicts and a smaller literal bound distance [12]. This holds an optimistic future for research in this field, as there is potential for a faster implementation that can achieve less conflicts.

The paper is organized as follows. Firstly, we talk about all of the preliminary knowledge rqeuired in Section 2. We explore all relevant related work in Section 3. The problem is elaborated further in Section 4, which will be followed by Section 5 for our contributions to this field. We later talk about the experimentation, including the results, in Section 6. We end the paper with our conclusions in Section 7.

2 Preliminaries

2.1 Notation

Since our main concern is with tasks, we first define what a task is. For each task i, it has:

- est_i: Earliest start time of i.
- lct_i: Latest completion time of i.
- p_i: Processing time of i.

These can later be extended to define more terms, such as:

- $lst_i = lct_i - p_i$: Latest start time of i.
- $ect_i = est_i + p_i$: Earliest completion time of i.

All of the previously stated variables can also apply for sets of tasks, as we will commonly be grouping up tasks. For a set of activities T, they would work as follows:

- $est_T = \min\{est_j, j \in T\}$
- $lct_T = \min\{lct_j, j \in T\}$
- $p_T = \sum_{j \in T} p_j$

However, calculating the the true ect and lst of T is an NP-hard problem by itself [16]. As such we will be using a lower bound:

- $ect_T = \max\{est_{T'} + p_{T'}, T' \subseteq T\}$
- $lst_T = \min\{lst_{T'} - p_{T'}, T' \subseteq T\}$

2.2 Constraint Satisfaction Problem

The idea of constraint programming can be written as a *constraint satisfaction problem* (CSP). A CSP is a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ which hold the set of variables $\mathcal{X}$, the set of domains $\mathcal{D}$ and the set of constraints $\mathcal{C}$. If any of the domains in $\mathcal{D}$ are empty or if any of the constraints are not satisfied, then the solution is *infeasible*.

Variables in constraint programming are coupled with domains, which dictate the possible values the variables can hold. Domains can be of different types, such as a boolean (represented as $\{0, 1\}$), integer or real number. For task scheduling problems, we will consider an integer domain which represent the starting time. In order to link a variable with a domain we use a function, $\phi : \mathcal{X} \rightarrowtail \mathcal{D}$, which converts a variable to its corresponding domain.

Constraints on the other hand is best thought of the way to relate variables with each other. There are many different kinds of constraints which offer different characteristics, for example `all-different` which forces all variables to be different. They can be mathematically written as a function which takes input the ϕ function and returns a boolean, or $c : \phi \rightarrowtail \{0, 1\}$. If the constraint returns a 0 then the constraint is not satisfied and the solution is invalid and vice versa.

Propagators are functions that filter the domains from infeasible values, as a result they can be thought of a function $p : \mathcal{D}^n \rightarrowtail \mathcal{D}^n$, where n is the number of variables. Since propagators prune the domain, the resulting $\mathcal{D}^n$ should contain subsets of the input domains, in other words the result $(\mathcal{D}'_1, \ldots, \mathcal{D}'_n) = p(\mathcal{D}_1, \ldots, \mathcal{D}_n)$ and $\forall i \in \mathcal{X} : \mathcal{D}'_i \subseteq \mathcal{D}_i$.

Lazy clause generation is used as a means to do smarter backtracking and better traversal of the search tree with the use of explanations [5]. Whenever a propagation or conflict occurs, we generate explanation as to the reason this occurred. An explanation is structured in a SAT problem [5] which describes the bounds of a variables. An example can be seen in Equation 1, where we set the lower bound of x_1 to be zero, upper bound of x_2 to be ten which propagated x_3 to be three.

$$[x_1 \geq 0] \wedge [x_2 \leq 10] \rightarrow [x_3 = 5] \tag{1}$$

Whenever we generate an explanation based off of a conflict, we generate a *nogood*. Nogoods have the same structure as an explanation but lead to an infeasibility, denoted as $\bot$. We can turn explanations into nogoods by negating the right-hand side of the explanation, for example we can change the explanation shown in Equation 1 to:

$$[x_1 \geq 0] \wedge [x_2 \leq 10] \wedge \neg[x_3 = 5] \rightarrow \bot \tag{2}$$

2.3 Not-First/Not-Last

We can now describe how the Not-First/Not-Last propagators work given the notation previously stated. In this case, we will focus primarily on the Not-Last propagator, as they are symmetrical.

The Not-Last propagtor looks at whether for a set of tasks $\Omega \subset T$ there can be a task $i \in (T \setminus \Omega)$ that can not be executed after Ω. For this to occur, the earliest completiton time of Ω must be greater than the latest possible start time of i, resulting in i not being able to start before its deadline:

$$lst_i < ect_\Omega \tag{3}$$

As such, at least one task from Ω must run after i. Since we know this is the case, lct_i can be updated to a value less than what it currently is:

$$lct'_i = \min\{lct_i, \max\{lst_j, j \in \Omega\}\} \tag{4}$$

We can rewrite this for all possible sets of tasks to complete the full propagator:

$$\forall \Omega \subset T, \forall i \in (T \setminus \Omega) : lst_i < ect_\Omega \Rightarrow lct'_i = \min\{lct_i, \max\{lst_j, j \in \Omega\}\} \tag{5}$$

Thanks to Torres and Lopez [14], we can prune T to remove any elements which will not cause propagation, based on the Equation 4. These tasks are the ones that start before i can be processed, as otherwise $\max\{lst_j, j \in \Omega\}$ in Equation 4 will be greater or equal to lct_i, which leads to no propagation. They defined this set as $\text{NLSet}(T, i)$:

$$\text{NLSet}(T, i) = \{j, j \in T \wedge lct_i > lst_j \wedge j \neq i\} \tag{6}$$

2.4 Conflict Windows

In our case, we decided to write explanations as conflict windows, which are introduced by Petr Vilím[1]. A conflict window for a task i is of the shape $\langle \underline{est_i}, \underline{lct_i} \rangle$, and it has a relaxed range of the task, meaning $\langle est_i, lct_i \rangle \subseteq \langle \underline{est_i}, \underline{lct_i} \rangle$. Infeasibility of a task no matter what arrangement can be shown with $\langle -\infty, \infty \rangle$.

As can be seen in Figure 1, there are two tasks that overlap and thus conflict. Conflict windows are generated which are show when they will still conflict given their current positions. Note that the conflict windows can be greater or equal to the *est* and *lct* of the tasks.

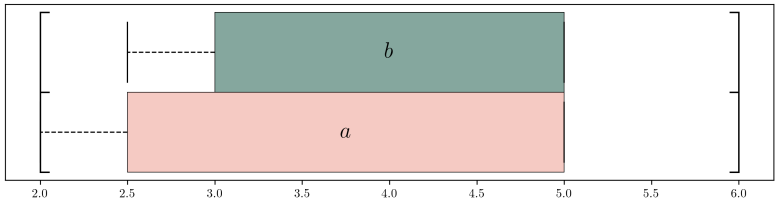

Figure 1 Two tasks a and b with square brackets indicating their conflict windows accordingly. In this case the conflict windows extend until there would still be a overlap of the two tasks.

In the case of Not-First/Not-Last, we must ensure that all inequalities still remain valid when defining the conflict windows. We first look at the Inequality 3 which states that the earliest completion time for a task is after the set of tasks. We then modify this as follows:

$$\begin{aligned} lst_i &< ect_\Omega \\ lct_i - p_i &< est_\Omega + p_\Omega \\ lct_i - p_i - p_\Omega &< est_\Omega \\ \forall j \in \Omega : \underline{est_j} &> lct_i - p_i - p_\Omega \end{aligned} \tag{7}$$

Likewise, for the assignment of the the latest completion time in Equation 4:

$$\begin{aligned} lct'_i &= \min\{lct_i, \max\{lst_j, j \in \Omega\}\} \\ lct'_i &\geq \max\{lst_j, j \in \Omega\} \\ lct'_i &\geq \max\{lct_j - p_j, j \in \Omega\} \\ \forall j \in \Omega : \underline{lct_j} &\leq lct'_i + p_j \end{aligned} \tag{8}$$

As such, we can change the conflict windows of all tasks in Ω:

$$\forall j \in \Omega : \langle lct_i - p_i - p_\Omega + 1, lct'_i + p_j \rangle \tag{9}$$

As for the conflict window for $i : \langle -\infty, lct_i \rangle$, since the task should be completed earlier than the initial lct_i.

After generating the conflict windows, we must convert them to an explanation. We can consider a conflict window for an arbitrary task x_n of $\langle a, b \rangle$, which states than it must start after a and end before b. This would be translated as $[x_n \geq a] \wedge [x_n \leq b]$, but if either limits are infinity then those can be ignored since they are always true. As such this leads to the following explanation for the Not-Last propagator:

$$\left(\bigwedge_{j \in \Omega} [j \geq \underline{est}_j] \wedge [j \leq \underline{lct}_j] \right) \rightarrow [i \leq lct_i] \tag{10}$$

3 Related Work

Constraint programming has come a long way, and has been proven to be useful for a variety of NP-Hard problems [8]. Task scheduling in particular has been shown to be NP-Hard [10], meaning that constraint programming can be applied. Since then, the cumulative and disjunctive constraints have developed propagators to effectively prune the domains of variables, with one of them being the Not-First/Not-Last propagators.

The first implementation was created by Baptise [3] while exploring the edge-finding propagator. This implementation had a time complexity of $\mathcal{O}(n^2)$ and space complexity of $\mathcal{O}(n)$.

During this time, explanations were introduced in task scheduling for Constraint Programming by Gueret [7], which allows for more intelligent searching of the search tree. While not directly implementing explanations for the Not-First/Not-Last propagators, they were a step towards incorporating disjunctive propagators with explanations in the solver.

Subsequently, the implementation by Torres and Lopez [14] only looked at the set of tasks that could affect propagation, which they named NFSet and NLSet accordingly. However their implementation involved doing a iterating through all tasks, and then checking for each one if a change can be made. As such, it kept the time and space complexity of $\mathcal{O}(n^2)$ and $\mathcal{O}(n)$ respectively.

Afterwards, it was further improved using a Θ tree by Petr Vilím[15]. This tree structure has a time complexity of $\mathcal{O}(\log n)$ for calculations of the earliest completion time of a set of tasks. Given the loop for all tasks, this results in a runtime of $\mathcal{O}(n \log n)$.

In due time, the first explanations for the Not-First/Not-Last propagators were developed by Petr Vilím[1]. He introduced the concept of conflict windows which indicate is a more relaxed range of the task's earliest start time and latest start time that still causes a conflict.

Finally, Lazy Clause Generation (LCG) solvers were discovered by Feydy [5]. LCG solvers only looked through explanations and generated learnt clauses only when a conflict arose. By moving the order of execution of the SAT solver to when a conflict occurred, it allows for different search strategies to be used.

In this paper, we will mix ideas from Petr Vilím's papers [1, 15] by expanding the conflict windows while also finding a smaller set. As Petr discussed in his paper "global constraints in scheduling" [16], there could be a subset of NLSet that can still propagate a task. In this paper, our goal is to find that set while also relaxing the range as much as possible to give more general explanations.

4 Problem Description

The known task scheduling problem [2] involves arranging the order of execution of tasks based on given information about each task. No matter what the task is, given *earliest start time*, *latest completion time* and *processing time*, constraint programming can be used to figure out whether solutions exist.

The disjunctive constraint in particular looks at execution of tasks in a a non-preemptive manner with one resource, meaning only one task can be executed at a time and not disrupted. With better observation of the tasks, we can shorten the earliest start times and latest completion times. In the Not-First/Not-Last case, we can group up tasks and figure out if a task can be put *before* or *after* this group, accordingly. If this is infeasible, then we can laten the earliest start time, or shorten the latest completion time.

In Lazy Clause Generation solvers, this can later be expanded through the use of explanations. Explanations can help with giving a reason for a propagation or conflict, and later generate learnt clauses. The more general of an explanation, meaning the less variables involved or wider ranges, the better as it has more use for learnt clauses. The final goal is to see the performance difference if we find a subset of the tasks which propagate the task, creating a more general explanation.

More formally written, given a set of tasks $T = \{t_1, \ldots, t_n\}$, with each task t being represented by a tuple (est_t, lct_t, p_t), the idea of the disjunctive constraint is to find an arrangement of tasks such that they do not overlap. We can filter out the domains of tasks via propagators, which in the case of tasks means increasing est_t or decreasing lct_t.

The Not-Last propagator work by picking a subset $\Omega \subset T$, and checking for a task $i \in (T \setminus \Omega)$ if $ect_\Omega > lst_i$. If that is the case, then we can update lct_i to be $\min\{lct_i, \max\{lst_j, j \in \Omega\}\}$. We can use the $\text{NLSet}(T, i) = \{j, j \in T \wedge lst_j < lct_i \wedge j \neq i\}$ to only use tasks which could affect the propagation of the variable. However since ect_{NLSet} is equal to $\max\{est_{\Omega'} + p_{\Omega'}, \Omega' \subseteq \text{NLSet}(T, i)\}$ we could attempt to find Ω' in order to have less tasks to use in the explanations, giving us more general explanations.

An example can be seen in Figure 2 which shows four tasks along with their *est* and *lct*. In this case, the task to be propagated is c, as such we have $\text{NLSet}(T, c) = \{a, b, d\}$, generating an explanation that involves three different tasks. Our goal would be to find a subset $\Omega' \subseteq \{a, b, d\}$ which will still cause propagation, however have a smaller size.

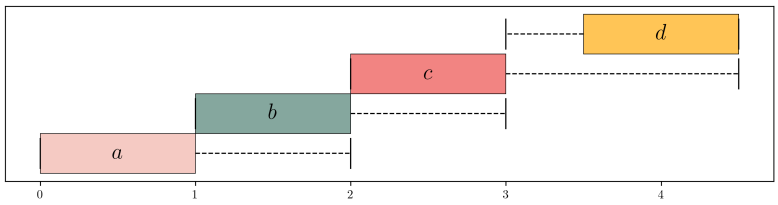

Figure 2 Figure showing four tasks a, b, c and d along with their corresponding *est* and *lct*.

5 Main Contributions

We will take steps on exploring the new ideas to finding better explanations. Firstly we will start by generating naive conflict windows for tasks per propagation. Afterwards, we will look into the method of calculating conflict windows for tasks by Petr Vilím[1]. Later on, we

will expand on it by using the conflict windows explained in Equation 9. And finally, we will look at subsets that lead to potentially better propagation and explanations.

Before we start looking into improvements, we need to look at the baseline explanations, which are the naive explanations. Naive explanations set conflict windows to be the current *est* and *lct* of all tasks at the moment of propagation. Meaning instead of Equation 9, we simply return $\forall j \in T : \langle ect_j, lct_j \rangle$.

Since our goal is to make explanations more general, we can also *only* look at sets of tasks that we know might cause propagation. In the case of Not-Last, this set of tasks is called the NLSet, first discovered by Torres and Lopez [14]. The idea of this set is that in Equation 4, we care about the smaller value between lct_i and $\max\{lst_j, j \in \Omega\}$. As such, we should look at the set of task Ω where $lst_j < lct_i$, otherwise $lst_j \geq lct_i$, and then lct_i would not get updated.

Naturally, naive conflict windows are not optimal, as such Petr looked into expanding the conflict window to the point where the tasks *still* cause a conflict. In the case of Not-Last, we can look at the Equations 7, 8 and 9. This expands the range of a conflict window compared to the naive explanations, creating more general explanations compared to the naive explanations.

In practice, not all of the tasks from the NLSet are required in order to find a propagation of a task. An example can be seen in Figure 3, where task c does not affect the propagation as it is really far away. By using the wider conflict windows proposed by Vilìm [1], we would also have to change the conflict window of c. By going through all of the subsets of NLSet, we can instead find the smallest set which still causes a propagation. Resulting in more general conflict windows, which will overall lead to less conflicts [1].

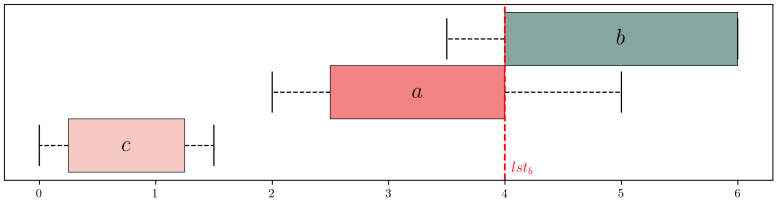

Figure 3 Three tasks are shown, a b and c, with their corresponding *est* and *lct*. A vertical line shows lst_b, which will be the propagted value of lct_a and also taken into consideration for the conflict windows.

Given a set of tasks, iterating through all subsets is a $\mathcal{O}(2^n)$ in time and space complexity, which can significantly slow down a solver. As such, we implemented Algorithm 1, which considers going through all subsets of a set *only* when their size is smaller than a constant k. This allows for the time and space complexity to change into $\mathcal{O}(2^k n \log n)$ for the propagators, where the increase is negligible for small numbers of k.

Algorithm 1 Search for smaller subset that propagates

Input: T: set of tasks, k: max size, i: task to be propagated
Output: $\overline{T}$: Smallest subset

```
1:  if len(T) > k then return T
2:  else
3:      lst ← ∞
4:      T ← ∅
5:      for T' in P(T) do
6:          lst' ← max{lst_j, j ∈ T'}
7:          if ect_T' > lst_i and lst' < lst then
8:              T ← T'
9:              lst ← lst'
10:         end if
11:     end for
        return T
12: end if
```

While heuristics could have been used in order to find the smallest set, we believed it was better to first explore all of the subsets. The main issue with using heuristics is that it is not trivial as to what the heuristic should be. This is due to the fact that we would like to find Ω' in $ect_\Omega = \max\{est_{\Omega'} + p_{\Omega'}, \Omega' \subseteq \Omega\}$, which does not necessarily mean that a greedy approach would find this answer. As such, figuring out ect_Ω is still NP-Hard, which consequently means our approach also can not be optimized.

6 Experiments

While the proposed methods are theoretically sound, we would like to see the difference that this would bring in practice. As such, we compare test instances for an implementation of the naive from Section 5, state-of-the-art from Section 2.4 and our explanation from Section 5 in order to compare the difference in time, amount of conflicts and average literal bound distance (LBD). We have found that around 50% of our test instances encounter less conflicts and a lower LBD, however almost all of them take longer to run. In addition, we have run our explanations on a maximum of $k = 5$ and saw how often our strategy ran and what difference would a higher value contribute in terms of conflicts, time and LBD needed to run. These resulted to our expected results of more tasks being considered with a higher k, which in returns takes longer to run, however the number of conflicts and LBD do not necessarily decrease.

6.1 Datasets

MinZinc[1] is a tool which allows us to control the solver, the model and the data. Thanks to minizinc, all tests will be ran on our solver called "Pumpkin"[2]. A standard minizinc model has been written which allows for a number of "machines" and tasks, depending on the data provided by the test instance, which result to the number of disjunctive constraints.

[1] `https://www.minizinc.org/`
[2] Forked version of Pumpkin which NF/NL was developed in: `https://github.com/dprin/Pumpkin`

Moreover, we used the VSIDS [11] search strategy, which is more likely to select tasks that have been found during conflict analysis.

We used two standard datasets that are often present in Petr Vilím's work [1, 16]. The two datasets we used are from Lawrence [9] and ORB dataset from the OR library [4]. We decided to run these datasets as not only is it widely used, but also features a wide variety of problems. They range from small instances with 10 jobs and 5 machines to significantly challenging with 15 jobs and 15 machines. In the end we used all of the dataset, meaning from LA01 to LA40 and ORB01 to ORB10, in order to rigorously test our propagators.

Additionally, we generated our own tests with Taillard's method [13] for more instances. We decided to create more tests to reflect on test cases that are capable of proving optimality and not timing out after a significant[3] amount of time. In our case, we have 10 instances with 10 jobs and 10 machines, and 10 instances with 15 jobs and 8 machines. The specific seeds used are shown in the Appendix B, in order for future researchers to reproduce the results.

In the end, we have a good dataset to benchmark different types of explanations. All of them were ran on an Intel i7-12700H at 4.7GHz clock speed in a HP Zbook Power G9. In order to make sure all types of explanations are treated fairly, we ran them with a timeout of 10 minutes and compared the shared lowest result. Moreover, datasets are grouped and averaged per 5 instances, this allows for a fair comparison between harder instances and easier ones.

6.2 Comparison of our method with naive and Petr Vilím's method

In order to comprehensively compare the different types of explanations, we must make sure we have fair metrics. We chose three metrics that will be of use to us, time required, amount of conflicts and literal block distance [12]. Time required is an important metric, which for most who want to schedule tasks are interested in the most. Amount of conflicts indicates the number of times an infeasible arrangement was encountered while finding optimal solution. Literal block distance [12] is a good indication of the difference between decision levels in learnt clauses, in our case a smaller number is better because our solver can negate more clauses at once, leading to less conflicts.

The time metric is shown in a scatter plot in Figure 4 and in the Table 1. There are instances such as ORB06-10, for $k = 3$ where the algorithm is faster in Table 1, and that is also reflected in the scatter plot in ranges 10-100. However the majority of times, the Petr Vilím's explanations find the optimal solution faster. Moreover, other than for instances LA01-15, it is always faster than the naive solution. We believe this is due to the added complexity in going through all subsets for each propagation while additionally the data collection for checking how many times we iterate through the powerset.

The amount of conflicts can be seen in the scatter plot in Figure 5 and in the Table 1. Throughout most cases, we can see that there is a decrease of amount of conflicts compared to the state-of-the-art. In addition, in all instances it is significantly smaller than the naive implementation. This was an expected result based on the fact that we have more general explanations which allow for better backtracking.

Finally, the avereage literal block distance can be seen in Figure 6 and in the Table 1. Compared to the state-of-the-art, overall there is a mixed result with around less than half

[3] We consider a significant amount of time to be the point where it would not provide an optimal result after 10 minutes as practically we could not prove optimality even after extending the timeout for longer.

of the instances having a smaller average literal block distance. Furthermore, it is always significantly less than the naive implementation. These results are due to the fact that the solver still mixes different learnt clauses from different decision trees, which does not improve this metric.

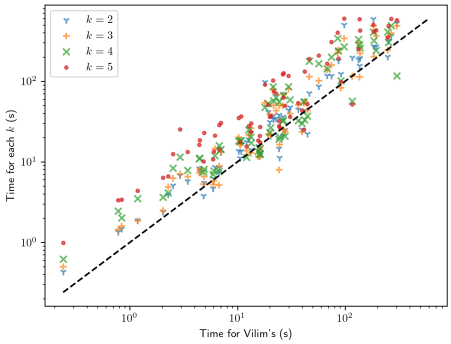

Figure 4 Scatter plot of time to required to find optimal solution for different values of k in our method compared to state-of-the-art implementation. Points below the line indicate less time than the state-of-the-art, while above indicate more.

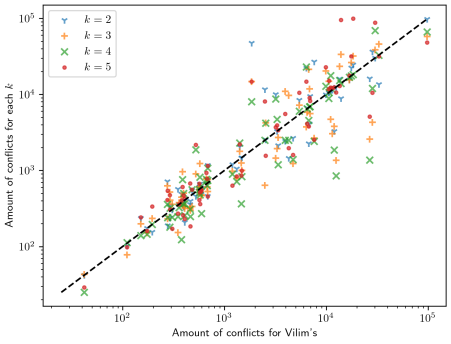

Figure 5 Scatter plot of the number of conflicts encountered while finding optimal solution for different values of k for our explanation compared to state-of-the-art implementation. Points below the line indicate less time than the state-of-the-art, while above indicate more.

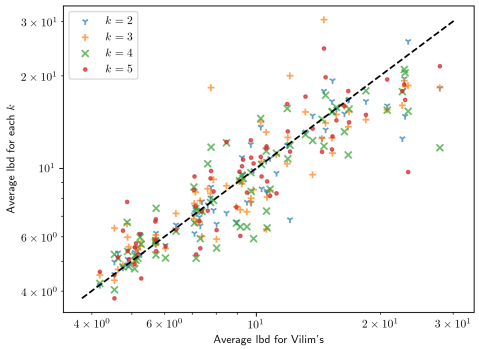

Figure 6 Scatter plot of the average literal block distance for different values of k for our explanation compared to state-of-the-art implementation. Points below the line indicate less time than the state-of-the-art, while above indicate more.

Table 1 Table comparing the naive, standard implementation and our new explanations with a maximum set size of three. Four metrics are shown, time in seconds, amount of conlficts and literal block distance [12] and the amount of times we explored the powerset in percent.

	naive				*Vilim's*				$k = 3$			
	time	conflicts	lbd	%used	time	conflicts	lbd	%used	time	conflicts	lbd	%used
la01-05	1	1.7K	10	0	1	355	6	0	2	257	6	30
la06-10	6	1.6K	18	0	7	302	7	0	12	295	6	20
la11-15	30	3.7K	22	0	24	492	8	0	40	595	10	15
la16-20	28	23.0K	16	0	5	3.0K	8	0	10	2.5K	7	31
la21-25	404	131.1K	28	0	48	12.7K	17	0	83	5.2K	15	20
la26-30	354	36.5K	45	0	147	7.7K	20	0	208	8.5K	20	15
la31-35	212	226	36	0	228	380	5	0	405	450	6	10
la36-40	530	77.8K	31	0	116	5.2K	15	0	243	7.6K	13	20
orb01-05	181	169.1K	16	0	30	14.7K	12	0	33	22.6K	11	29
orb06-10	112	89.4K	16	0	25	11.9K	10	0	19	15.1K	10	30
ta_j10m10_01-05	19	9.2K	16	0	7	1.0K	6	21	7	901	6	31
ta_j10m10_06-10	13	5.1K	13	0	10	831	5	21	12	1.1K	6	31
ta_j15m8_01-05	171	118.1K	22	0	42	24.5K	12	13	58	16.3K	12	19
ta_j15m8_06-10	180	114.8K	22	0	25	11.0K	13	13	45	11.7K	12	20

6.3 Best performing maximum set limit

Allowing for bigger sets of tasks prove to give better conflict windows but at the cost of an increase in time complexity. In our case, we could run up to a maximum size of five, and after that the speed decrease was significant. We used four metrics:

- amount of time required
- amount of conflicts
- average literal block distance
- amount of times we iterate through all subsets

As is visible in Table 2, on average we see a better time for a maximum size of four due to two factors. The first one being that we have to iterate through less subsets compared to a higher value such as five. In addition, the more general explanations allow for better traversal of the search tree.

Moreover, we see a smallest number of conflicts when the maximum size is four in Table 2. It was expected to be a larger number, since we would be giving more general explanations for more propagations. The cause of this is due to the fact that a different search path by VSIDS was taken which lead to more conflicts.

Furthermore, the average literal block distance was collected and we received some surprising results. There is no clear *best* value for the maximum size of the set, which is counter-intuitive as we expect a larger number to correlate with better explanations. This is due to the fact that we are mixing different strategies depending on the size of the sets of tasks that we encounter.

Finally, we see a trend of iterating through subsets more often the higher the value of k. This was an expected result, as by increasing the value of k we encounter more sets of tasks which can be looked further into.

Table 2 Table containing different maximum size of sets to be considered, from two to five. Four metrics are shown, time in seconds, amount of conlficts and literal block distance [12] and the amount of times we explored the subsets in percent.

	$k = 2$				$k = 3$				$k = 4$				$k = 5$			
	time	conflicts	lbd	%used	time	conflicts	lbd	%used	time	conflicts	lbd	%used	time	conflicts	lbd	%used
la01-05	2	333	6	20	2	257	6	30	2	350	6	40	4	278	5	51
la06-10	11	269	6	13	12	295	6	20	10	254	7	26	14	299	7	33
la11-15	44	523	8	10	40	595	10	15	52	538	8	19	57	540	9	24
la16-20	9	2.9K	7	21	10	2.5K	7	31	12	1.9K	7	42	22	3.4K	7	53
la21-25	89	24.6K	13	13	83	5.2K	15	20	80	9.3K	16	27	98	9.5K	17	34
la26-30	249	10.1K	19	10	208	8.5K	20	15	285	4.9K	15	20	425	7.1K	17	25
la31-35	397	450	5	7	405	450	6	10	329	331	5	13	481	443	6	16
la36-40	201	5.2K	12	14	243	7.6K	13	20	256	6.9K	13	27	352	4.6K	14	34
orb01-05	26	16.2K	11	20	33	22.6K	11	29	26	14.7K	12	40	73	50.4K	11	49
orb06-10	15	9.4K	11	20	19	15.1K	10	30	20	11.9K	10	40	31	13.6K	10	51
ta_j10m10_01-05	7	1.0K	6	21	7	901	6	31	9	1.3K	6	42	16	1.2K	7	53
ta_j10m10_06-10	10	831	5	21	12	1.1K	6	31	15	1.1K	6	42	25	618	6	53
ta_j15m8_01-05	59	24.5K	12	13	58	16.3K	12	19	67	16.2K	10	26	76	16.5K	13	33
ta_j15m8_06-10	37	11.0K	13	13	45	11.7K	12	20	59	18.2K	13	26	76	22.6K	13	33

7 Conclusion

Ultimately, we explored explanations for the Not-First/Not-Last propagator. We did this by looking through all of the subsets of the set of tasks that affect propagation, and using the smallest set that still propagated. In addition, we considered using this method only when the set is less than a specified maximum.

We found that overall it takes more time to find an optimal solution for test instances using our method compared to Petr Vilím's explanations [1]. However we discovered that half of the time we have less conflicts and a lower average of literal block distance.

For future work, it is recommended to look into if there it is possible to use heuristics for decreasing the size of tasks, rather than looking through all subsets. Moreover, higher values of k need to be explored to see the change in conflicts and average literal block distance.

A Responsible Research

There are primarily two factors to consider when working on a propagator, reproducibility and access to test instances. They are important factors to keep in mind when working on research, as they allow for faster progression in research.

Everything written in the paper is available in a public GitHub repository[4]. This allows for future researchers to look into our code and figure out the inner workings in practice, rather than theoretically.

Finally, most test instances that were ran are widely used and available in this field. They are publicly available in the well known OR library [4], and we also made sure to cite all

[4] `https://www.github.com/dprin/pumpkin`

papers from which the datasets originate from. In addition, we generated our own data which is available on the GitHub repository[5], while also including the seeds used in Appendix B.

A.1 AI Usage

During the research of this project LLMs have been used to assist during data processing. They were primarily used to make the tables in the paper, by converting them from a Python dictionary to a Pandas table. Additionally, they were used during the creation of the examples in the paper, in which the code was later manually modified to fit our needs.

B Seeds used for custom TA tests

Custom seeds are used when generating the test instances that can be seen in the Tables 1, 2 and 4.

Filename	Time Seed	Machine Seed
instance_j10m10_1.dzn	595312809	1226873846
instance_j10m10_2.dzn	1994010092	674268132
instance_j10m10_3.dzn	684069210	17711631
instance_j10m10_4.dzn	433393242	1856683078
instance_j10m10_5.dzn	1737541870	2123655630
instance_j10m10_6.dzn	1053901157	2093288715
instance_j10m10_7.dzn	197486053	2072736806
instance_j10m10_8.dzn	541533994	836141230
instance_j10m10_9.dzn	591728687	1250988364
instance_j10m10_10.dzn	64076732	175007812
instance_j15m8_1.dzn	166140137	21143938
instance_j15m8_2.dzn	472453410	1957606298
instance_j15m8_3.dzn	325953502	1085494782
instance_j15m8_4.dzn	417269198	833500955
instance_j15m8_5.dzn	1001380920	1585883286
instance_j15m8_6.dzn	607390163	1104152727
instance_j15m8_7.dzn	1561051784	2003679251
instance_j15m8_8.dzn	422129019	1697884185
instance_j15m8_9.dzn	1213865971	957331870
instance_j15m8_10.dzn	1039225514	117613006

Table 3 Table containing all seeds used when generating Taillard tests [13]. The first column represents the file name, second column the time seed and the third the machine seed.

C All experimental data

[5] https://www.github.com/dprin/pumpkin

	$Vilim's$				$k=2$				$k=3$				$k=4$				$k=5$			
	time	conflicts	lbd	%used	time	conflicts	lbd	%used	time	conflicts	lbd	%used	time	conflicts	lbd	%used	time	conflicts	lbd	%used
la01	1	110	5	0	2	102	5	20	2	78	6	30	3	113	5	40	4	98	5	51
la02	1	348	7	0	1	570	8	20	1	153	6	30	2	249	7	41	3	272	7	51
la03	1	685	7	0	1	461	7	19	2	529	8	29	2	741	6	38	3	457	6	50
la04	2	591	5	0	2	488	5	20	2	480	5	30	4	621	5	40	6	535	5	51
la05	0	42	5	0	0	42	5	20	0	43	4	30	1	25	4	40	1	29	4	51
la06	3	151	6	0	6	237	6	14	7	198	5	20	8	140	5	27	13	243	6	33
la07	3	407	10	0	5	208	8	13	6	336	8	19	8	365	8	26	12	426	11	32
la08	12	463	6	0	18	398	6	13	14	301	6	20	11	248	7	25	16	184	5	31
la09	2	195	9	0	4	155	7	13	5	235	8	20	4	195	7	26	7	336	7	33
la10	13	293	5	0	23	349	6	13	26	407	6	20	20	320	6	26	24	308	6	32
la11	27	565	7	0	69	602	7	10	68	563	7	15	66	515	7	19	126	671	7	24
la12	24	566	7	0	38	608	7	10	50	613	8	15	55	562	7	19	32	411	8	24
la13	19	480	11	0	42	553	8	10	22	336	8	15	53	558	6	20	51	559	9	25
la14	20	277	7	0	21	189	6	10	22	232	8	15	26	241	7	20	37	544	9	24
la15	31	574	8	0	47	665	9	10	40	1.2K	18	15	59	814	12	19	36	516	10	24
la16	4	4.7K	8	0	8	2.7K	7	21	8	1.2K	6	31	11	1.4K	6	42	19	1.6K	6	53
la17	3	291	5	0	7	239	5	21	7	522	6	32	11	180	5	42	25	476	5	53
la18	4	1.3K	6	0	8	1.1K	7	21	7	792	7	31	11	718	6	42	16	838	6	52
la19	7	5.4K	9	0	13	8.4K	11	20	13	7.2K	9	31	14	6.0K	10	42	31	10.5K	10	53
la20	6	3.4K	9	0	9	2.1K	8	21	15	2.7K	7	31	12	1.2K	6	42	21	3.4K	8	52
la21	46	6.4K	21	0	71	22.3K	16	13	114	3.7K	15	20	177	23.1K	15	27	189	14.7K	19	34
la22	37	26.6K	16	0	44	16.2K	15	13	51	2.6K	16	20	32	1.4K	13	27	54	5.1K	16	34
la23	21	522	8	0	37	459	7	14	80	899	9	20	86	1.9K	12	27	102	2.2K	12	34
la24	116	28.3K	23	0	197	36.4K	15	13	116	4.3K	18	20	57	12.0K	20	27	52	10.6K	17	33
la25	18	1.8K	15	0	98	47.4K	13	13	54	14.8K	14	20	48	8.0K	17	27	91	14.8K	20	34
la26	248	3.9K	15	0	201	2.4K	18	10	243	11.0K	30	15	411	2.5K	12	20	352	5.6K	25	25
la27	98	7.6K	23	0	501	26.9K	26	10	342	2.7K	19	15	269	2.4K	15	20	599	2.5K	10	25
la28	68	13.5K	23	0	118	16.0K	20	10	139	23.3K	19	14	143	15.3K	21	20	165	12.9K	19	25
la29	136	11.9K	28	0	156	3.3K	18	10	113	3.8K	18	15	284	1.9K	12	20	592	12.4K	21	25
la30	185	1.4K	10	0	270	2.0K	14	10	203	1.8K	14	15	320	2.3K	15	20	414	2.2K	11	25
la31	253	461	5	0	317	358	5	7	356	319	5	10	302	310	5	13	485	675	6	16
la32	182	275	5	0	593	716	5	7	488	629	5	10	407	360	5	13	502	410	5	17
la33	133	392	6	0	186	356	6	7	298	355	6	10	329	364	6	13	251	258	6	17
la34	268	396	5	0	364	330	5	7	395	502	6	10	492	500	5	13	594	470	5	17
la35	303	377	5	0	526	489	6	7	488	444	6	10	117	123	5	13	575	402	6	16
la36	74	3.2K	14	0	116	10.1K	15	14	159	1.5K	10	20	199	8.7K	12	27	309	3.7K	13	34
la37	138	11.0K	18	0	193	11.3K	17	14	222	4.7K	14	21	259	17.7K	18	27	428	12.3K	15	34
la38	85	686	11	0	245	1.2K	8	14	227	762	6	21	341	1.1K	7	28	401	1.2K	12	35
la39	194	4.3K	12	0	270	1.5K	7	14	366	9.7K	20	20	241	2.5K	12	27	430	2.0K	11	35
la40	87	6.7K	23	0	181	2.3K	12	14	243	21.4K	16	20	239	4.5K	18	27	192	3.8K	18	34
orb01	26	13.9K	14	0	21	8.8K	13	20	50	33.5K	13	28	32	13.9K	14	39	123	95.1K	11	47
orb02	16	6.9K	9	0	16	9.4K	9	20	12	5.6K	9	31	14	6.9K	9	41	18	8.2K	10	52

Continued on next page

	$Vilim's$				$k=2$				$k=3$				$k=4$				$k=5$			
	time	conflicts	lbd	%used	time	conflicts	lbd	%used	time	conflicts	lbd	%used	time	conflicts	lbd	%used	time	conflicts	lbd	%use
orb03	42	18.3K	15	0	44	24.5K	17	19	56	31.9K	13	28	33	18.3K	15	39	132	99.2K	12	46
orb04	28	17.5K	9	0	25	22.4K	10	19	24	26.2K	9	29	21	17.5K	9	40	50	31.7K	11	50
orb05	40	16.8K	10	0	25	15.7K	9	20	24	15.5K	11	30	30	16.8K	10	40	43	17.6K	9	50
orb06	44	32.9K	15	0	22	13.6K	19	19	43	45.9K	11	28	37	32.9K	15	40	52	31.6K	13	50
orb07	16	10.7K	8	0	13	10.8K	8	20	12	11.4K	8	30	12	10.7K	8	40	21	15.1K	9	51
orb08	24	2.5K	10	0	15	11.6K	12	18	8	639	8	30	19	2.5K	10	40	28	8.1K	12	50
orb09	16	6.8K	8	0	12	4.1K	7	20	15	11.8K	9	30	13	6.8K	8	41	27	9.0K	8	52
orb10	25	6.5K	11	0	11	6.6K	10	20	16	5.8K	11	30	20	6.5K	11	40	27	4.1K	8	51
ta_j10m10_01	6	593	7	21	5	593	7	21	5	467	7	31	7	272	5	42	11	555	5	53
ta_j10m10_02	5	387	5	21	4	387	5	21	5	966	7	31	7	759	7	42	10	607	8	53
ta_j10m10_03	7	666	6	21	7	666	6	21	5	776	6	31	8	763	5	42	14	932	7	53
ta_j10m10_04	6	306	5	21	6	306	5	21	6	359	6	32	8	242	6	43	13	172	4	53
ta_j10m10_05	13	3.3K	7	21	14	3.3K	7	21	13	1.9K	7	31	17	4.7K	9	42	30	3.9K	9	53
ta_j10m10_06	11	2.5K	8	21	11	2.5K	8	21	16	4.2K	9	31	18	4.2K	10	42	18	1.6K	7	53
ta_j10m10_07	5	174	4	21	6	174	4	21	7	152	5	32	8	145	4	42	23	161	5	53
ta_j10m10_08	7	408	5	21	8	408	5	21	9	318	5	31	16	257	5	41	29	230	5	52
ta_j10m10_09	11	358	5	21	13	358	5	21	17	378	5	32	18	326	5	42	37	358	6	53
ta_j10m10_10	16	676	6	21	13	676	6	21	13	606	6	32	15	633	6	42	17	776	7	53
ta_j15m8_01	30	9.9K	11	13	44	9.9K	11	13	80	20.3K	13	19	86	12.9K	11	26	117	22.8K	12	33
ta_j15m8_02	27	12.5K	17	12	36	12.5K	17	12	25	1.4K	14	20	33	850	11	27	63	10.6K	18	33
ta_j15m8_03	41	1.5K	10	13	58	1.5K	10	13	53	1.3K	9	20	56	366	6	27	25	862	10	34
ta_j15m8_04	91	97.9K	13	11	123	97.9K	13	11	83	57.9K	15	18	101	66.4K	13	24	98	48.1K	17	31
ta_j15m8_05	22	575	9	13	34	575	9	13	50	548	7	20	57	479	6	27	75	361	6	34
ta_j15m8_06	57	30.1K	17	13	86	30.1K	17	13	102	38.6K	15	19	165	69.0K	16	25	209	87.8K	14	31
ta_j15m8_07	20	10.6K	16	13	31	10.6K	16	13	39	14.9K	15	19	23	8.9K	16	26	36	11.7K	14	33
ta_j15m8_08	23	11.5K	12	13	34	11.5K	12	13	43	3.0K	10	20	63	11.6K	16	26	70	12.1K	16	33
ta_j15m8_09	13	1.2K	11	13	19	1.2K	11	13	21	978	8	20	25	896	9	27	28	633	8	33
ta_j15m8_10	10	1.5K	12	13	14	1.5K	12	13	20	1.0K	13	20	18	834	11	26	35	993	13	34

References

1 Computing Explanations for the Unary Resource Constraint. In *Lecture Notes in Computer Science*, pages 396–409. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISSN: 0302-9743, 1611-3349. URL: http://link.springer.com/10.1007/11493853_29, doi:10.1007/11493853_29.

2 David Applegate and William Cook. A Computational Study of the Job-Shop Scheduling Problem. *ORSA Journal on Computing*, 3(2):149–156, May 1991. URL: https://pubsonline.informs.org/doi/10.1287/ijoc.3.2.149, doi:10.1287/ijoc.3.2.149.

3 Philippe Baptiste and Claude Le Pape. Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling. Citeseer, 1996.

4 J. E. Beasley. OR-Library: Distributing Test Problems by Electronic Mail. *Journal of the Operational Research Society*, 41(11):1069–1072, November 1990. URL: https://www.tandfonline.com/doi/full/10.1057/jors.1990.166, doi:10.1057/jors.1990.166.

5 Thibaut Feydy and Peter J. Stuckey. Lazy Clause Generation Reengineered. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, volume 5732, pages

352–366. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/978-3-642-04244-7_29, doi:10.1007/978-3-642-04244-7_29.

6 Keivan Ghoseiri, Ferenc Szidarovszky, and Mohammad Jawad Asgharpour. A multi-objective train scheduling model and solution. *Transportation Research Part B: Methodological*, 38(10):927–952, December 2004. URL: https://linkinghub.elsevier.com/retrieve/pii/S0191261504000098, doi:10.1016/j.trb.2004.02.004.

7 Christelle Guéret, Narendra Jussien, and Christian Prins. Using intelligent backtracking to improve branch-and-bound methods: An application to Open-Shop problems. *European Journal of Operational Research*, 127(2):344–354, December 2000. URL: https://linkinghub.elsevier.com/retrieve/pii/S0377221799004889, doi:10.1016/S0377-2217(99)00488-9.

8 Jena-Lonis Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29–127, February 1978. URL: https://linkinghub.elsevier.com/retrieve/pii/0004370278900292, doi:10.1016/0004-3702(78)90029-2.

9 Lawrence, S. Resource constrained project scheduling : an experimental investigation of heuristic scheduling techniques (Supplement). 1984. URL: https://cir.nii.ac.jp/crid/1571980073974705920.

10 J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of Machine Scheduling Problems. In *Annals of Discrete Mathematics*, volume 1, pages 343–362. Elsevier, 1977. URL: https://linkinghub.elsevier.com/retrieve/pii/S016750600870743X, doi:10.1016/S0167-5060(08)70743-X.

11 Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers. In Nir Piterman, editor, *Hardware and Software: Verification and Testing*, volume 9434, pages 225–241. Springer International Publishing, Cham, 2015. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/978-3-319-26287-1_14, doi:10.1007/978-3-319-26287-1_14.

12 Laurent Simon and Gilles Audemard. Predicting Learnt Clauses Quality in Modern SAT Solver. In *Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*, Pasadena, United States, July 2009. URL: https://inria.hal.science/inria-00433805.

13 E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, January 1993. URL: https://linkinghub.elsevier.com/retrieve/pii/037722179390182M, doi:10.1016/0377-2217(93)90182-M.

14 Philippe Torres and Pierre Lopez. On Not-First/Not-Last conditions in disjunctive scheduling. *European Journal of Operational Research*, 127(2):332–343, December 2000. URL: https://linkinghub.elsevier.com/retrieve/pii/S037722179900497X, doi:10.1016/S0377-2217(99)00497-X.

15 Petr Vilím. O(nlog n) Filtering Algorithms for Unary Resource Constraint. In Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Jean-Charles Régin, and Michel Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3011, pages 335–347. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. Series Title: Lecture Notes in Computer Science. URL: https://link.springer.com/10.1007/978-3-540-24664-0_23, doi:10.1007/978-3-540-24664-0_23.

16 Petr Vilím. Global constraints in scheduling. 2007. Publisher: Univerzita Karlova, Matematicko-fyzikální fakulta.