Cumulative Scheduling with Calendars and Overtime

Samuel Cloutier
Université Laval, Québec, Canada

Claude-Guy Quimper
Université Laval, Québec, Canada

Abstract

In project scheduling, calendar considerations can increase the duration of a task when its execution overlaps with holidays. On the other hand, the use of overtime may decrease the task's duration. We introduce the CALENDAROVERTIME constraint which verifies that a task follows a calendar with overtime and holidays. We also introduce the CUMULATIVEOVERTIME constraint, a variant of the CUMULATIVE constraint, that also reasons with the calendars when propagating according to the resource consumption, the overtime, and the holidays. Experimental results of a RCPSP model on the PSPLIB, BL, and PACK instances augmented with calendars and overtime show that the use of the CALENDAROVERTIME constraint offers a speedup greater than 2.9 on the instances optimally solved and finds better solutions on more than 79% of the remaining instances when compared to a decomposition of the constraint. We also show that the use of our CUMULATIVEOVERTIME constraint further improves these results.

2012 ACM Subject Classification Computing methodologies → Planning and scheduling; Theory of computation → Constraint and logic programming

Keywords and phrases Constraint programming, Scheduling, Global constraints, Calendars, Overtime, Cumulative constraint, Time-Tabling

Digital Object Identifier 10.4230/LIPIcs.CP.2024.7

Supplementary Material *Software (Source Code and Instances)*: `https://github.com/Samclou/chuffed/releases/tag/Calendars-cp2024`

1 Introduction

In project management, it is common to schedule a variety of tasks on a project timeline. With multiple machines and workers, some tasks can easily be done in parallel. For example, a furniture factory can build a table at the same time as a chair, as long as sufficient workers and workspace are available. Cumulative scheduling allows the simultaneous execution of tasks while limiting these executions in order not to overload the resources.

Scheduling problems (with release times and deadlines) are generally NP-hard [9]. Constraint programming is frequently used to solve these problems.

In practice, tasks can be suspended for some time periods. The factory can be closed at night and during weekends. If every operation is stopped at these times, these time periods can simply be ignored. If some tasks must be stopped at specific times while others do not because, for example, the machines keep working at night, side constraints become necessary to encode these suspensions and this may undermine the efficiency of the models.

The aim of this research is to design constraints that facilitate the modeling and solving of scheduling problems where tasks must be interrupted according to a calendar, or may be shortened by working overtime.

Section 2 provides background on the cumulative scheduling problem, the Time-Tabling rule, and the generalizations with calendars. Section 3 presents the new constraints we introduce. Section 4 describes how these constraints can be decomposed into elementary constraints while Section 5 details the propagators of the new constraints. Section 6 describes the methodology we used to test our new propagators. Section 7 evaluates the performances of the new propagators and the decomposition.

2 Background

2.1 Cumulative Scheduling

The cumulative scheduling problem is often modeled with the CUMULATIVE global constraint [1]. In what follows, lower-case symbols represent constants and indices while upper-case ones represent variables. Symbols in bold represent arrays that we define using list comprehension. Let $\mathcal{I}$ be the set of tasks and let p_i, h_i, and S_i for $i \in \mathcal{I}$ be the task's processing time, usage of a resource, and starting time. Let $\mathbf{S}$ be $[S_i \mid i \in \mathcal{I}]$, $\mathbf{p}$ be $[p_i \mid i \in \mathcal{I}]$, and $\mathbf{h}$ be $[h_i \mid i \in \mathcal{I}]$. The constraint CUMULATIVE$(\mathbf{S}, \mathbf{p}, \mathbf{h}, h^{\max})$ asserts that for a resource of capacity $h^{\max}$ that executes the tasks in $\mathcal{I}$, for any integer time point t in the *horizon* (the complete time interval considered, $[0, t^{\max}]$), $\sum_{i \in \mathcal{I}: t \in [S_i, S_i+p_i)} h_i \leq h^{\max}$. This means that tasks running simultaneously cannot, at any time, consume more than the resource's capacity. The execution window of a task is considered to be $[S_i, S_i + p_i)$, with $S_i + p_i$ being its ending time. As such, S_i takes integer values in $[0, t^{\max} - p_i]$. The CUMULATIVE constraint uses filtering algorithms to prune the variable domains during the search. Since enforcing bounds consistency is NP-hard [17], one usually applies simple filtering rules that offer a weaker level of consistency such as the Time-Tabling rule [2]. In a multi-resource problem, each resource is associated with its own CUMULATIVE constraint. A common objective is to minimize the *makespan*, i.e., the completion time of the last task.

Lazy clause generation [18] is a technique that deduces new logical constraints, in the form of a disjunction of literals representing domain states of the variables, from the failures encountered during the search. It permits learning previous bad decisions and prune them from the remaining search tree. Solvers that implement lazy clause generation, such as Chuffed [8], have been shown to perform well on cumulative scheduling problems [20].

2.2 The Time-Tabling Rule

Let $\underline{X}$ and $\overline{X}$ respectively be the smallest and largest values a variable X can take. We note $\mathrm{dom}(X)$ the set of all values variable X may take, i.e., its domain. The Time-Tabling rule [2] filters the domains of the starting time variables subject to the CUMULATIVE constraint. We note the earliest starting time of task i as est_i, its latest starting time as lst_i, its earliest completion time as ect_i, and its latest completion time as lct_i. These are defined as follows:

$$\mathrm{est}_i \overset{def}{=} \underline{S_i} \tag{1}$$

$$\mathrm{lst}_i \overset{def}{=} \overline{S_i} \tag{2}$$

$$\mathrm{ect}_i \overset{def}{=} \underline{S_i} + p_i \tag{3}$$

$$\mathrm{lct}_i \overset{def}{=} \overline{S_i} + p_i \tag{4}$$

If a task duration is a variable, these definitions use the lower bound of that variable rather than p_i. These four concepts bound the time points at which a task can be in execution. A task must be in execution in the interval $[\mathrm{lst}_i, \mathrm{ect}_i)$, called the *compulsory part*, if it is non-empty.

The Time-tabling rule computes the compulsory part of each task and aggregates them to create a *consumption profile*, i.e., a lower bound of the resource consumption at each time point. The Time-Tabling check identifies a conflict when a point in this profile overloads the resource. The Time-Tabling filtering algorithm makes sure that if a task overloads the resource when executing at time t, then the task must either start after or finish before t [19]. Let $f(\Omega, t)$ be the consumption profile of a resource at time t given the tasks in the set Ω.

$$f(\Omega, t) = \sum_{\{i \in \Omega \mid t \in [\mathrm{lst}_i, \mathrm{ect}_i)\}} h_i \tag{5}$$

The checking and filtering rules for the cumulative constraint can then be expressed as:

$$\exists\, t, f(\mathcal{I}, t) > h^{\max} \implies \text{conflict} \tag{6}$$

$$\mathrm{ect}_i > t \wedge h^{\max} < h_i + f(\mathcal{I} \setminus \{i\}, t) \implies \underline{S_i'} > t \tag{7}$$

Rule (7) can be adapted to filter $\overline{S_i}$. Propagators applying the Time-Tabling rule can have a complexity as low as $\mathcal{O}(n)$, n being the number of tasks. However, there exist efficient implementations with a complexity of $\mathcal{O}(n^2)$ [10].

2.3 Augmentation With Calendars

It is possible that, at some specific times, some tasks must be paused while others remain unaffected. We say that these special times are defined by a *calendar*. This notion is close to preemption, but it is still in a non-preemptive context. A task can only be suspended because of calendars and it must resume as soon as each calendar affecting the task permit it. There are multiple ways to conceptualize calendars and many ways to solve the problem have been studied.

2.3.1 Calendars Associated to Resources

One way to add calendars into the cumulative scheduling problem is to assign to each resource an arbitrary array of Booleans indicating whether the resource is available or not at a specific time. When a resource is unavailable, tasks cannot progress in their execution, which has the effect of artificially lengthening their execution time.

Kreter et al. [12, 13, 14] use *releasable resources* that stop being consumed by tasks that are paused. Their tasks may have an initial uninterruptible setup time. To deal with their complex problem, Kreter et al. study various methods:

- They use multiple binary linear model formulations and search methods that they compare against each other [12].
- They implement a new constraint, namely CUMULATIVECALENDAR, in a constraint solver and compare its efficiency with various models using existing constraints [13].
- They compare both previous methods on the resource investment problem, i.e., the problem of minimizing the cost associated to the maximum consumption of each resources [14].

Kreter et al. [13, 14] show that the use of CUMULATIVECALENDAR constraints with a lazy clause generation solver such as Chuffed is highly competitive to solve their problems.

2.3.2 Calendars Associated to Tasks

Boudreault et al. [6] directly assigns the calendars to the tasks, meaning that each task follows its own calendar, rather than following one implied by those of the resources. This might be wanted over the preceding option when some resources are plentiful enough that modeling them with a Cumulative constraint would be useless. However, if these omitted resources have a calendar, they still need to affect the actual calendar of a task, which justifies using this more general type of calendar. For Boudreault et al., the calendars are not arbitrary as the composition of a working day is fixed and shared between all calendars: the regular execution time starts at a given time in the morning and finishes at a given time in the afternoon. Some tasks can execute during the weekend while other tasks cannot. The calendars are periodic on weeks and do not allow exceptions even for holidays.

Boudreault et al. [6] allow overtime, i.e., a way to shorten the execution time of a task while inducing an overtime cost. Working one time point worth of overtime on task i costs w_i. The amount of overtime is limited by its availability. Indeed, overtime is assigned to specific time points in the calendar, during which a task can be interrupted or can continue its execution if it is executed in overtime.

To solve their problem, Boudreault et al. [6] do not implement a new constraint in a solver, they rather decompose the calendar constraints into elementary constraints available in any constraint solver and use a meta-heuristic to reach better results.

2.3.3 Other Approaches

In CP Optimizer, tasks in scheduling problems are modeled through *interval* variables. These variables possess a starting time and an ending time, but also a *size*, and a step function, called *intensity*. The size of an interval variable can be interpreted as the work contained in the interval, while the intensity gives the ratio of work that each time point provides. As such, the behavior that tasks do not progress during holidays can directly be treated through the intensity function by having an intensity of 0% during that time. If tasks are not allowed to start or end during holidays, constraints forbidStart and forbidEnd directly model and deal efficiently with this aspect [15].

Beldiceanu [4, 5] introduces a Calendar constraint to model this behavior. This constraint maps, for each calendar, the real-time coordinate system to a virtual one where there are no interruptions. These virtual time coordinates then permit the use of classic propagators that normally cannot deal with calendars. The mapping deals with the problem of changing the length of tasks and of making sure none starts nor ends during a holiday.

3 Calendar Constraints With Overtime

This section presents the new constraints we introduce. The next section presents how they can be decomposed while the following section describes their filtering algorithms.

The constraints we introduce are motivated by the calendar constraints used by Boudreault et al. [6]. We generalize the calendars they use by allowing arbitrary calendars, i.e., non-periodic calendars with sporadic holidays. Every task must follow a specific calendar. The *elapsed time* of a task is the difference between its end time and its start time. Without calendars, the elapsed time of a task is simply its processing time.

Given a horizon representing all the time points at which a task can be processed, our calendars are arbitrary sequences of the symbols $\mathbb{r}$, $\mathbb{c}$, or $\mathbb{o}$ where the t-th symbol represents the nature of the t-th time point, i.e., the t-th hour in our context. The symbol $\mathbb{r}$ indicates that the time point is regular (the classic scheduling problem would correspond to a calendar with only $\mathbb{r}$'s). The symbol $\mathbb{c}$ indicates that the time point is closed, that is, tasks are suspended when they are in process at that time. As for $\mathbb{o}$, it indicates that the time point is an overtime period that can behave as a regular or a closed time point whether it is worked or not. The duration that a task is worked in overtime is the number of time points of type

$\mathbb{o}$ that behave as type $\mathbb{r}$. The time point of the start and the one preceding the end of a task must not be closed. If either of these points is an overtime period, the amount of worked overtime must allow to work them. The time worked in the execution window of a task must be exactly its processing time.

▶ **Example 1.** Let i be a task with processing time $p_i = 3$ following the calendar $\mathbb{coroorrc}$. This task cannot start at time 0 since it is closed. It can execute at times 1, 2, and 3. It could also execute at times 1, 2, and 4. In that case, it finishes later and is idle at time 3. The task cannot start at time 1 while ending at time 7 because the regular times 2, 5, and 6 are mandatory, leaving no work to perform at time 1.

3.1 The CalendarOvertime Constraint

We define a new constraint to model calendars with overtime. The CalendarOvertime constraint, for a starting time variable S, an elapsed time variable E (with $\text{dom}(E) \subseteq [p, t^{\max}]$), an overtime variable O (with $\text{dom}(O) \subseteq [0, p]$), a processing time p, and a calendar Cal, asserts that:

- The first and last time points of the execution window $[S, S+E)$ are not closed.
- There are enough, but not too many, worked time points in the window to complete the task of processing time p with the overtime prescribed by O.
- There are enough overtime periods in $[S, S+E)$ for the overtime prescribed by O.
- The first and last time points in $[S, S+E)$ can actually be worked if they are of type $\mathbb{o}$.

In a more mathematical way, the CalendarOvertime constraint is defined as follows:

$$\begin{aligned}
&\textsc{CalendarOvertime}(S, E, O, p, \text{Cal}) \overset{def}{\iff} \text{Cal}[S] \neq \mathbb{c} \wedge \text{Cal}[S+E-1] \neq \mathbb{c} \\
&\qquad \wedge O = p - |\{t \in [S, S+E) \mid \text{Cal}[t] = \mathbb{r}\}| \\
&\qquad \wedge O \leq |\{t \in [S, S+E) \mid \text{Cal}[t] = \mathbb{o}\}| \\
&\qquad \wedge |\{t \in \{S, S+E-1\} \mid \text{Cal}[t] = \mathbb{o}\}| \leq O
\end{aligned}$$

This constraint does not deal with the concept of resource consumption. It simply maintains consistency between the variables S, E, and O given a processing time p and a calendar Cal.

3.2 The CumulativeOvertime Constraint

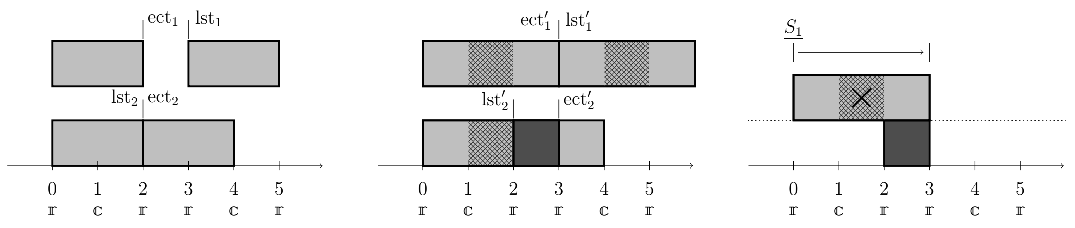
Let $\mathbf{E}$ be $[E_i \mid i \in \mathcal{I}]$, $\mathbf{O}$ be $[O_i \mid i \in \mathcal{I}]$, and $\mathbf{Cal}$ be $[\text{Cal}_i \mid i \in \mathcal{I}]$. We define the new CumulativeOvertime constraint as follows:

$$\begin{aligned}
&\textsc{CumulativeOvertime}(\mathbf{S}, \mathbf{E}, \mathbf{O}, \mathbf{p}, \mathbf{Cal}, \mathbf{h}, h^{\max}) \\
&\overset{def}{\iff} \textsc{Cumulative}(\mathbf{S}, \mathbf{E}, \mathbf{h}, h^{\max}) \\
&\qquad \wedge \bigwedge_{i \in \mathcal{I}} \textsc{CalendarOvertime}(S_i, E_i, O_i, p_i, \text{Cal}_i)
\end{aligned}$$

In words, we define the new CumulativeOvertime constraint as a conjunction of a Cumulative constraint and the CalendarOvertime constraints associated to the tasks consuming the resource. Importantly, we consider that tasks continue to use the resource while they are suspended, be it because of unworked overtime or closed time. This last aspect is reasonable if we consider that some resources may relate to small spaces where moving machinery should be avoided, such as during ship refitting. In that case, releasing the resource necessitates unwanted work that would overcomplicate the planning.

Although the propagator for CalendarOvertime (described at Section 5.1) maintains bounds consistency on S, E, and O given processing time p and calendar Cal, the bounds found on E are often not sufficient to allow the Cumulative constraint to perform a good propagation. This is because the Time-Tabling rule filtering the Cumulative constraint only uses $\underline{E}$ in its reasoning and does not take into account the calendars.

▶ **Example 2.** Consider a task 1 of processing time $p_1 = 2$ following the calendar $\mathtt{rcrrcr}$ with $\mathrm{dom}(S_1) = [0, 3]$ and $\mathrm{dom}(E_1) = [2, 3]$. Let task 2 follow the same calendar, with $p_2 = 2$, $\mathrm{dom}(S_2) = [0, 2]$ and $\mathrm{dom}(E_2) = [2, 3]$. In this case, $O_1 = O_2 = 0$. It can be seen that the domains of the variables of both tasks are bounds consistent with respect to the CalendarOvertime constraints and that they do not induce any compulsory part. Let both tasks consume 1 unit of a resource of capacity 1 (i.e., a disjunctive resource). Then, the Time-Tabling rule is not able to deduce that S_1 should be fixed to 3. Indeed, not only does it fail to detect that task 2 must be executing at time 2 (meaning its compulsory part in Figure 1a should not be empty), it also considers that task 1 could start at time 0 and end at time 2 (excluded), leading to no propagation. Should the calendar considerations be included in the rule, the propagation would be deduced, as visible in Figures 1b and 1c.



(a) When not considering the calendar, the time windows that end the earliest or start the latest have no intersection. The profile is empty and no propagation is possible.

(b) When taking the calendar into account, the tasks execution windows become longer, allowing the detection of compulsory parts that would otherwise be missed.

(c) With this augmented profile, calendar considerations can again lengthen a task's execution window and permit the detection of propagations that would be missed otherwise.

Figure 1 Comparison of the propagation done by the Time-tabling rule without and with calendar considerations in the case described by Example 2. The meaning of ect'_i and lst'_i is defined at section 5.1.

4 Decomposition of the New Constraints

To evaluate the usefulness of the new constraints defined in the previous section, we need to compare them with their decomposition into elementary constraints.

4.1 Decomposition of the CalendarOvertime Constraint

Suppose we want to decompose the constraint CalendarOvertime$(S, E, O, p, \mathrm{Cal})$. If Cal is the trivial calendar, i.e., the calendar with only regular time points without closed time or overtime, the constraint is trivially decomposed as follows:

$$E = p \wedge O = 0 \tag{8}$$

If Cal is not the trivial calendar, more work is necessary. Let the *compiled calendars* $C^{\mathbb{c}}$, $C^{\mathbb{r}}$, and $C^{\mathbb{o}}$ respectively count how many closed, regular, and overtime periods are encountered in calendar Cal before a given time point in the horizon $[0, t^{\max}]$. The number of closed time points in the time interval $[a, b)$ is simply given by $C^{\mathbb{c}}[b] - C^{\mathbb{c}}[a]$. These arrays can be precomputed.

$$C^{\mathbb{x}}[t] = |\{j \in [0, t) \mid \mathrm{Cal}[j] = \mathbb{x}\}| \qquad \forall \mathbb{x} \in \{\mathbb{c}, \mathbb{r}, \mathbb{o}\} \tag{9}$$

The following variables are added to the decomposition: I, for the idle time i.e., the number of time points in $[S, S+E)$ that are not worked, as well as $\mathrm{N}^{\mathbb{c}}$, $\mathrm{N}^{\mathbb{r}}$, and $\mathrm{N}^{\mathbb{o}}$ respectively for the number of closed, regular, and overtime time points within the execution window $[S, S+E)$.

First, variables $\mathrm{N}^{\mathbb{c}}$, $\mathrm{N}^{\mathbb{r}}$, and $\mathrm{N}^{\mathbb{o}}$ must count the time points of each type in the execution window.

$$\mathrm{N}^{\mathbb{x}} = C^{\mathbb{x}}[S+E] - C^{\mathbb{x}}[S] \qquad \forall \mathbb{x} \in \{\mathbb{c}, \mathbb{r}, \mathbb{o}\} \tag{10}$$

The number of regular time points in the execution window must be equal to the regular time worked. There must be enough overtime periods in the execution window to work the overtime prescribed by O. The idle time is not only the closed time points, but also the unworked overtime periods. Since overtime periods appear directly in the calendar, the elapsed time is simply the processing time plus the idle time. The overtime is nonnegative and at most equal to the processing time.

$$\mathrm{N}^{\mathbb{r}} = p - O \tag{11}$$

$$\mathrm{N}^{\mathbb{o}} \geq O \tag{12}$$

$$I = \mathrm{N}^{\mathbb{c}} + \mathrm{N}^{\mathbb{o}} - O \tag{13}$$

$$E = p + I \tag{14}$$

$$0 \leq O \leq p \tag{15}$$

The starting time and the time preceding the ending time of a task must be able to be worked, even if they are overtime periods. Let $\mathbb{1}(x)$ be the function that returns 1 if x is true and 0 otherwise.

$$\mathrm{Cal}[S] \neq \mathbb{c} \tag{16}$$

$$\mathrm{Cal}[S+E-1] \neq \mathbb{c} \tag{17}$$

$$E > 1 \implies O \geq \mathbb{1}(\mathrm{Cal}[S] = \mathbb{o}) + \mathbb{1}(\mathrm{Cal}[S+E-1] = \mathbb{o}) \tag{18}$$

$$E = 1 \implies O \geq \mathbb{1}(\mathrm{Cal}[S] = \mathbb{o}) \tag{19}$$

4.2 Decomposition of the CumulativeOvertime Constraint

A decomposition of the constraint CUMULATIVEOVERTIME$(\mathbf{S}, \mathbf{E}, \mathbf{O}, \mathbf{p}, \mathbf{Cal}, \mathbf{h}, h^{\max})$ can simply consist of the constraint CUMULATIVE$(\mathbf{S}, \mathbf{E}, \mathbf{h}, h^{\max})$ along with the decomposition of CALENDAROVERTIME$(S_i, E_i, O_i, p_i, \mathrm{Cal}_i)$ for each $i \in \mathcal{I}$ given by constraints (8) to (19).

5 Filtering Algorithms for the New Constraints

The strength of the new constraints over their decomposition, aside from the modeling simplification they bring, is the stronger propagation they permit. This is possible thanks to the filtering rules and algorithms presented in this section.

5.1 Propagation of the CalendarOvertime Constraints

For each task i, let $V_i(s,e,o)$ be a predicate satisfied if task i can start at time s for a duration of e with overtime o given the calendar Cal_i.

$$V_i(s,e,o) \overset{def}{\iff} s+e \leq \text{horizon} \wedge \textsc{CalendarOvertime}(s,e,o,p_i,\mathrm{Cal}_i) \tag{20}$$

Since Cal_i affects task i, est_i, lst_i, ect_i, and lct_i are redefined as follows:

$$\mathrm{est}'_i \overset{def}{=} \min\left\{s \in \left[\underline{S_i}, \overline{S_i}\right] \mid \exists e \in \left[\underline{E_i}, \overline{E_i}\right], \exists o \in \left[\underline{O_i}, \overline{O_i}\right],\, V_i(s,e,o)\right\} \tag{21}$$

$$\mathrm{lst}'_i \overset{def}{=} \max\left\{s \in \left[\underline{S_i}, \overline{S_i}\right] \mid \exists e \in \left[\underline{E_i}, \overline{E_i}\right], \exists o \in \left[\underline{O_i}, \overline{O_i}\right],\, V_i(s,e,o)\right\} \tag{22}$$

$$\mathrm{ect}'_i \overset{def}{=} \min\left\{s+e \mid s \in \left[\underline{S_i}, \overline{S_i}\right],\, e \in \left[\underline{E_i}, \overline{E_i}\right],\, o \in \left[\underline{O_i}, \overline{O_i}\right],\, V_i(s,e,o)\right\} \tag{23}$$

$$\mathrm{lct}'_i \overset{def}{=} \max\left\{s+e \mid s \in \left[\underline{S_i}, \overline{S_i}\right],\, e \in \left[\underline{E_i}, \overline{E_i}\right],\, o \in \left[\underline{O_i}, \overline{O_i}\right],\, V_i(s,e,o)\right\} \tag{24}$$

We consider that $\min(\emptyset) = \infty$ and $\max(\emptyset) = -\infty$.

Using definition (21), the checking and filtering rules for $\underline{S_i}$ in the CalendarOvertime propagator are:

$$\mathrm{est}'_i = \infty \implies \text{conflict} \tag{25}$$

$$\underline{S_i} < \mathrm{est}'_i \implies \underline{S'_i} = \mathrm{est}'_i \tag{26}$$

The filtering algorithm for the constraint CalendarOvertime is based on four precomputed vectors: Let $k \subseteq \{\mathbb{r}, \mathbb{o}\}$, be the types of time points the vectors consider. $C_i^k[t]$ is the number of time points of type in k that come before time t in Cal_i, and $Y_i^k[j]$ is the index of the j-th time point of type in k in Cal_i. With these vectors, we define helper functions that execute in constant time. For simplicity's sake, we only present sketches that ignore boundary conditions at the beginning or the end of the scheduling horizon. The function $\mathrm{count}_k(a,b) := C_i^k[b] - C_i^k[a]$ returns the number of time points in the time window $[a,b)$ with a type in k. $\mathrm{previous}_k(t) := Y_i[C_i^k[t+1]-1]$ returns the latest time point with a type in k that is not later than t. $\mathrm{next}_k(t) := Y_i[C_i^k[t]]$ returns the earliest time point with type in k that is not earlier than t. $\mathrm{get\_end}_k(t,\Delta) := Y_i[C_i^k[t]+\Delta]$ returns the end of the smallest time window beginning at t and containing Δ time points with a type in k. Finally,

$$\begin{aligned}\mathrm{verify\_head\_tail}(s,e) :=& \mathbb{1}(\mathrm{Cal}_i[s] = \mathbb{o}) + \mathbb{1}(e>1)\mathbb{1}(\mathrm{Cal}_i[s+e-1] = \mathbb{o}) \\ &\leq p_i - \mathrm{count}_{\{\mathbb{r}\}}(s, s+e)\end{aligned}$$

is true if and only if the time worked regularly in $[s, s+e)$ permits enough overtime to work in overtime on the first and last time points.

This constraint requires a constant number of variables per task. Because the vectors C_i^k and Y_i^k must be precomputed, the space complexity of the filtering algorithm is linear with respect to the horizon, and the initialization (performed once when instantiating the model) is also linear. Algorithm 1 computes in constant time a candidate value for ect'_i, as redefined by (23), given a fixed starting time s for a task i subject to a calendar. Algorithm 2 verifies the value given by Algorithm 1, and filters the lower bound of S_i according to the CalendarOvertime constraint. This algorithm iterates on $\mathrm{dom}(S_i)$, computing a minimal completion time for each candidate start time. The first start time leading to a finite completion time is the new lower bound. Even though the running time complexity is in $\mathcal{O}(|\mathrm{dom}(S_i)|)$, it is technically linear w.r.t. the number of filtered-out unclosed time points. As such, the algorithm runs in constant time if it filters nothing and it runs in linear time if it filters many values. The upper bound of S_i and the other variables are processed similarly.

Algorithm 1 Computing ect'_i given calendar Cal_i, and $S_i = s$.

```
1  Function compute_completion_time($i$, $s$):
2      if $\mathrm{Cal}_i[s] = \mathbb{c}$ then return $\infty$ ;
       // The execution window contains at least $p_i$ unclosed periods.
3      $\mathrm{end} \leftarrow \mathrm{get\_end}_{\{\mathbb{r},\mathbb{o}\}}(s, p_i)$
4      if $\mathrm{end} - s < \underline{E_i}$ then
           // The associated elapsed time must be at least $\underline{E_i}$.
5          $\mathrm{end} \leftarrow \mathrm{next}_{\{\mathbb{r},\mathbb{o}\}}(s + \underline{E_i} - 1) + 1$
6      $\mathrm{worked\_regular\_time} \leftarrow \mathrm{count}_{\{\mathbb{r}\}}(s, \mathrm{end})$
7      $\mathrm{min\_worked\_regular\_time} \leftarrow p_i - \overline{O_i}$
8      if $\mathrm{worked\_regular\_time} < \mathrm{min\_worked\_regular\_time}$ then
           // At least $p_i - \overline{O_i}$ regular time must be worked.
9          $\mathrm{end} \leftarrow \mathrm{get\_end}_{\{\mathbb{r}\}}(s, \mathrm{min\_worked\_regular\_time})$
10     if $\mathrm{Cal}_i[\mathrm{end} - 1] = \mathbb{o} \wedge \text{not verify\_head\_tail}(s, \mathrm{end})$ then
11         if $\mathrm{next}_{\{\mathbb{r}\}}(\mathrm{end} - 1) + 1 \leq \mathrm{horizon}$ then
               // Adding a regular time point fixes the tail problem.
12             $\mathrm{end} \leftarrow \mathrm{next}_{\{\mathbb{r}\}}(\mathrm{end} - 1) + 1$
       /* The ending time is minimal.  Constraints on $\overline{E_i}$, $\underline{O_i}$ or head and
          tail cannot be made right if they are not already.                        */
13     if $\mathrm{end} \leq \mathrm{horizon} \wedge \mathrm{end} - s \leq \overline{E_i} \wedge p_i - \mathrm{count}_{\{\mathbb{r}\}}(s, \mathrm{end}) \geq \underline{O_i}$
        $\wedge \mathrm{verify\_head\_tail}(s, \mathrm{end})$ then
14         return end
15     return $\infty$
```

Algorithm 2 iterates using naive unit leaps (see line 7). By analyzing the cause of why Algorithm 1 returns infinity, these leaps can be extended. For example, if the current s is a time point of type $\mathbb{r}$ and the failure is due to "end-s" at line 13 in Algorithm 1 being greater than $\overline{E}$ by k, then the "+1" in the leap could be replaced by a "+k". We have tested such enhancements but found no improvement on the performance. As such, the simpler version presented is the one used for the experimentations presented in Section 6.

5.1.1 Explaining the Propagation

In a solver with lazy clause generation, we explain propagations by rules (25) and (26) naively, respectively by (27) $\rightarrow$ False and (28) $\rightarrow \llbracket \mathrm{est}'_i \leq S_i \rrbracket$.

$$\llbracket \underline{S_i} \leq S_i \rrbracket \wedge \llbracket S_i \leq \overline{S_i} \rrbracket \wedge \llbracket \underline{E_i} \leq E_i \rrbracket \wedge \llbracket E_i \leq \overline{E_i} \rrbracket \wedge \llbracket \underline{O_i} \leq O_i \rrbracket \wedge \llbracket O_i \leq \overline{O_i} \rrbracket \tag{27}$$

$$\llbracket \underline{S_i} \leq S_i \rrbracket \wedge \llbracket \underline{E_i} \leq E_i \rrbracket \wedge \llbracket E_i \leq \overline{E_i} \rrbracket \wedge \llbracket \underline{O_i} \leq O_i \rrbracket \wedge \llbracket O_i \leq \overline{O_i} \rrbracket \tag{28}$$

Our previous attempts indicate that computing more general explanations is of little interest for this propagator compared to using the naive ones.

Algorithm 2 Filtering $\underline{S_i}$ given a calendar.

```
   Input: Variables $S_i$, $E_i$, and $O_i$.
1  $s \leftarrow \text{next}_{\{\mathtt{r},\mathtt{o}\}}(\underline{S_i})$
2  while $s \leq \overline{S_i}$ do
3      end $\leftarrow$ compute_completion_time($i$, $s$)
       // We only need to verify that it is a valid value for $\text{ect}'_i$.
4      if end $\neq \infty$ then
5          $\underline{S_i} \leftarrow s$
6          return Success
7      $s \leftarrow \text{next}_{\{\mathtt{r},\mathtt{o}\}}(s+1)$
8  return Conflict
```

5.2 Propagation of the CumulativeOvertime Constraints

The basis of the CumulativeOvertime propagator is that of a Cumulative propagator applying the classic Time-Tabling rule. The main difference is that it uses the definitions (21) to (24), rather than (1) to (4), to compute the profile with (5) and apply the Time-Tabling rules (6) and (7). Thus, $f'(\Omega, t) = \sum_{\{i \in \Omega \mid t \in [\text{lst}'_i, \text{ect}'_i)\}} h_i$ and the new checking and filtering rules are as follows:

$$\exists\, t, f'(\mathcal{I}, t) > h^{\max} \implies \text{conflict} \tag{29}$$

$$\text{ect}'_i > t \wedge h^{\max} < h_i + f'(\mathcal{I} \setminus \{i\}, t) \implies \underline{S'_i} > t \tag{30}$$

For that, the algorithm that enforces the Time-Tabling can compute the value ect'_i by calling Algorithm 1 (and verifying the value returned) with increasing values of $s \in \text{dom}(S_i)$. The first valid value returned is the ect'_i. The lst'_i is computed symmetrically. Most propagators applying the Time-Tabling rule can be adapted for the CumulativeOvertime propagator. Because of the computing time caused by the new definitions, the complexity of the CumulativeOvertime propagator is that of its base Cumulative propagator multiplied by the size of the largest domain of the starting time variables. Since we chose to adapt the propagator by Schutt et al. [20] that has a complexity of $\mathcal{O}(n^2)$, we obtain a propagator in $\mathcal{O}(kn^2)$, where $k = \max_{i \in \mathcal{I}} |\text{dom}(S_i)|$. Under the assumption that this constraint is used alongside CalendarOvertime constraints, the size of the scope of the constraint is the same as for the Cumulative constraint (here linear in the number of tasks).

This global propagator is used in combination with the propagators for the CalendarOvertime constraints. This is done because filtering the calendar constraints solely through this global propagator specialized for resource consumption would be inefficient.

5.2.1 Explaining the Propagation

In a solver using lazy clause generation, the propagation needs to be explained. First, should the propagator fail to find a valid ect'_i at some point in its execution, it means that the CalendarOvertime constraint cannot be satisfied. The CumulativeOvertime propagator directly reports a conflict that it naively explains with (27). As such, the rest of this section considers that est'_i, lst'_i, ect'_i, and lct'_i are valid. Let $t \in [\text{lst}'_i, \text{ect}'_i)$ be a time point in the calendar-corrected compulsory part of task i. The expression profile_expl is used to construct the explanation.

$$\text{profile\_expl}(i,t) \overset{def}{=} \begin{cases} [\![S_i \leq t]\!] \wedge [\![t+1-\underline{E_i} \leq S_i]\!] & \text{if } t \in \left[\overline{S_i}, \underline{S_i}+\underline{E_i}\right) \\ \quad \wedge [\![\underline{E_i} \leq E_i]\!] & \\ \quad (27) & \text{otherwise} \end{cases} \tag{31}$$

The expression profile_expl(i,t) depends on whether the redefinitions (21) to (24) are necessary to detect t as part of the compulsory part of task i. If the original definitions are sufficient, the explanation for t being in the compulsory part of task i is the same as presented by Schutt et al. [20], but with a variable duration. Otherwise, the explanation cannot be as general and we simply reuse the naive one presented previously.

Suppose that rule (29) finds a conflict at time t. We define $B_t \subseteq \mathcal{I}$ the set of tasks for which t is in their corrected compulsory part, i.e., $B_t = \{i \in \mathcal{I} \mid t \in [\text{lst}'_i, \text{ect}'_i)\}$. Let $B_t^* \subseteq B_t$ be a minimal set (in terms of number of elements) such that $\sum_{i \in B_t^*} h_i > h^{\max}$. Let t^+ be the smallest ect$'_i$ or lst$'_i$ greater than t and let t^- be the greatest ect$'_i$ or lst$'_i$ smaller than t. This means that every time point in the interval $[t^-, t^+]$ have the same set of tasks that have a compulsory part overlapping it, i.e., $B_t = B_{t'}$ for all $t' \in [t^-, t^+]$. As such, explaining based on any point in this interval is valid. Then, the propagator explains the conflict by:

$$\bigwedge_{i \in B_t^*} \text{profile\_expl}\left(i, \left\lfloor \frac{t^- + t^+}{2} \right\rfloor\right) \rightarrow \text{False}. \tag{32}$$

This corresponds to saying that the conflict is caused by a minimal number of tasks all having a compulsory part that includes the time point in the middle of the profile rectangle that contains t. If the calendar corrections (the new definitions (21) to (24)) are never needed, this explanation is the same as the *pointwise* explanation from Schutt et al.

For a task i and a time $t \in \left[\underline{S_i}, \text{ect}'_i\right)$, we define task_expl$(i,t)$ as follows:

$$\text{task\_expl}(i,t) \overset{def}{=} \begin{cases} [\![t+1-\underline{E_i} \leq S_i]\!] \wedge [\![\underline{E_i} \leq E_i]\!] & \text{if } t \in \left[\overline{S_i}, \underline{S_i}+\underline{E_i}\right) \\ \quad (28) & \text{otherwise} \end{cases}. \tag{33}$$

The logical expression task_expl(i,t) depends on whether calendar corrections are needed to detect that task i, when starting at a time not earlier than time $\underline{S_i}$, is not finished by time t. If so, we use a naive explanation like for the CalendarOvertime constraint. Otherwise, we reuse the expression from Schutt et al. [20].

Suppose that rule (30) pushes $\underline{S_i}$ to time $t+1$ and that t is the earliest time for which the rule applies. Let $B_t^* \subseteq B_t \setminus \{i\}$ be a minimal set such that $\sum_{k \in B_t^*} h_k > h^{\max} - h_i$. Then, the propagator instead filters $\underline{S_i}$ to $t^* = \min\{\text{ect}'_i, t^+\}$ and explains it by:

$$\text{task\_expl}(i,\ t^*-1) \wedge \bigwedge_{k \in B_t^*} \text{profile\_expl}\,(k,\ t^*-1) \rightarrow [\![t^* \leq S_i]\!]. \tag{34}$$

Rule (30) is reapplied until it no longer filters. This cuts the propagation from rule (30) into sub-propagations permitting, according to Schutt et al. [20], more general explanations. If the calendar corrections are never needed, these explanations are the same as the ones presented by Schutt et al.

6 Experimentation

To compare the value of our new propagators with the decomposition, we solve the following RCPSP model augmented with calendars and overtime.

6.1 Experimentation Model

The model has initial constraints on the time window of each task, task precedence constraints, resources that tasks need, and calendars that tasks follow. Let $\mathcal{R}$ be the set of resources and $\mathcal{I}$ a set of tasks. Each task $i \in \mathcal{I}$ has to start in a window $[\text{minStart}_i, \text{maxStart}_i]$ and end in a window $[\text{minEnd}_i, \text{maxEnd}_i]$. These windows encode release times and deadlines. Let $\mathcal{P} \subseteq \mathcal{I} \times \mathcal{I}$ contain the precedence relationships. For each $(i, j) \in \mathcal{P}$, the task i must end before the task j may start. The release times and deadlines provide the initial domains of the variables through the propagation of the following constraints:

$$\text{minStart}_i \leq S_i \leq \text{maxStart}_i \qquad \forall i \in \mathcal{I} \tag{35}$$

$$\text{minEnd}_i \leq S_i + E_i \leq \text{maxEnd}_i \qquad \forall i \in \mathcal{I} \tag{36}$$

The following constraints enforce the precedence relationships:

$$S_i + E_i \leq S_j \qquad \forall (i, j) \in \mathcal{P} \tag{37}$$

Finally, CumulativeOvertime constraints prevent the overload of the resources.

$$\textsc{CumulativeOvertime}(\mathbf{S}, \mathbf{E}, \mathbf{O}, \mathbf{p}, \mathbf{Cal}, \mathbf{h}_j, h_j^{\max}) \qquad \forall j \in \mathcal{R} \tag{38}$$

We either minimize the makespan (39) or the overtime costs (40):

$$\max_{i \in \mathcal{I}} \{S_i + E_i\} \tag{39}$$

$$\sum_{i \in \mathcal{I}} w_i O_i \tag{40}$$

We optimize these objective functions separately, i.e., optimizing only one function or the other. When minimizing the makespan, all overtime is forbidden. Otherwise, it would also maximize the overtime, which makes little sense for an applied project, since it leads to cost maximization.

By modifying how constraint (38) is implemented, we define three equivalent models:

- The CumulativeOvertime model implements constraint (38) directly with our global CumulativeOvertime constraint.
- The CalendarOvertime model decomposes constraint (38) with a classic Cumulative constraint, and a CalendarOvertime constraint for each task.
- The decomposition model decomposes constraint (38) as described in section 4.2.

6.2 Experimentation Details

We implement[1] the CalendarOvertime and CumulativeOvertime constraints in C++ in the solver Chuffed 0.13.0[2] [8], and write our models in MiniZinc [16]. To keep the comparison with the CumulativeOvertime model fair, the propagator that filters the Cumulative constraints in the CalendarOvertime and decomposition models only uses the Time-Tabling check and filtering already implemented in Chuffed. We run all experiments with a timeout of 10 minutes on a machine with a 32-core Intel Xeon 4110 CPU @ 2.10 GHz and 32 Gb of memory. We run four executions simultaneously, which may affect the precision of the runtimes.

[1] Available at: `https://github.com/Samclou/chuffed/releases/tag/Calendars-cp2024`
[2] Available at: `https://github.com/chuffed/chuffed/releases/tag/0.13.0`

We use the instances j30, j60, j90, and j120 from the PSPLIB [11] benchmark, the instances bl20 and bl25 from the BL set [3], and the PACK [7] instances, all adapted with randomly generated calendars where time points represent hours. The instances use calendars similar to those of Boudreault et al. [6], where days have 8 regular hours, followed by 4 hours of overtime. Some calendars have weekends off, and some do not have overtime. We add for each day a 5% chance for it to be a holiday. There is a calendar where weekends and holidays are composed of 12 overtime hours. These 2135 augmented instances and the models (as well as the execution logs) are accessible in the code repository.

For makespan minimization, we extend the horizons from the original instances by a factor of 5 to prevent the addition of closed hours from leading to trivial unsatisfiable instances. In these executions, we forbid overtime. For overtime costs minimization, we must use a smaller horizon to prevent having too many instances where the best value of 0 overtime is trivial to find, but it should not be reduced so much that we get easy unsatisfiable instances. To fix the horizon, we solve the instances twice to minimize the makespan: once by forbidding overtime and a second time by allowing overtime. We fix the horizon to the mean makespan. This gives a horizon for which there is always a solution, which is often not trivial and leaves room to optimize the overtime costs. The computation time required to compute these horizons is not taken into account in our results as they are used to construct the instances rather than solving the problem.

7 Results

Comparisons are made between the decomposition and the CalendarOvertime models, and between the CalendarOvertime and the CumulativeOvertime models.

7.1 Comparing the Decomposition and CalendarOvertime Models

Figure 2 shows graphs comparing the runtimes of our models on instances for which the solver proved the optimality. Compared to the decomposition model, the CalendarOvertime model represents an average speedup of 13.8 for makespan optimization and 2.9 for overtime optimization, respectively, on these 1625 and 1373 instances.

This speedup for the makespan optimization is larger than the one for the overtime optimization. We surmise that this important discrepancy is due to the size of the horizon in the makespan optimization instances. Indeed, their horizon is often very high compared to the optimal makespan or the horizon of the overtime optimization instances. It so happens that the Element constraints present in the decomposition of the CalendarOvertime constraint are susceptible to the size of the horizon. For example, in the solver used, constraint (16) becomes a collection of clauses that may each be as long as the horizon while constraint (10) is filtered by a propagator that is linear in the size of the horizon (as long as S and E are not fixed). This leads to both weak and slow filtering which must degrade the performances of the decomposition. We can see that the performances of the decomposition model become more competitive in the context of overtime minimization, which uses a tighter horizon.

Regarding the instances not solved optimally by both models, there are no instances where the decomposition model is able to prove optimality or find a solution better than the CalendarOvertime model. The CalendarOvertime model proves optimality on 17% of the 510 makespan instances and 8% of 762 overtime instances. It finds better solutions in 81% (79%) of makespan (overtime) instances. There are 21 instances for which the decomposition model fails to find any solution while the CalendarOvertime model is able to.

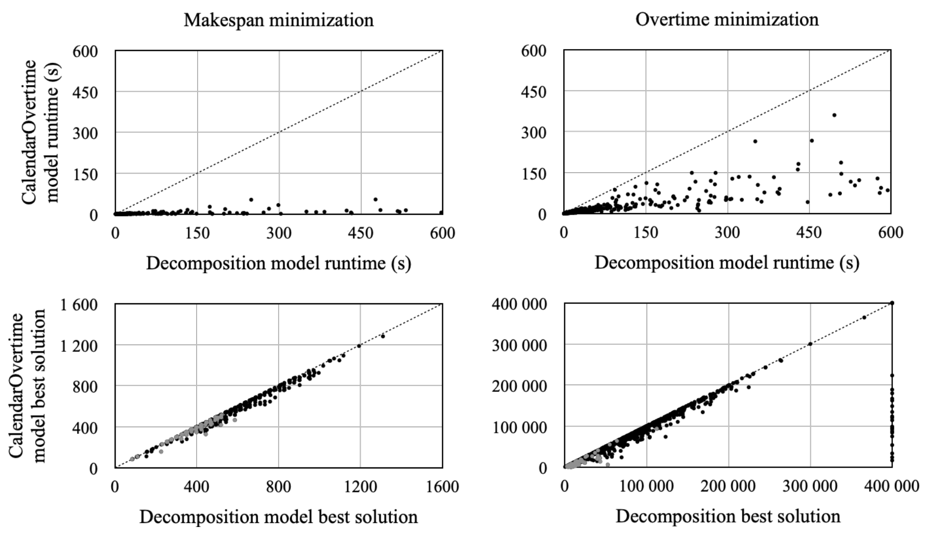


Figure 2 Comparison, between the decomposition and CalendarOvertime models, of the runtime on the instances solved by both models (1^{st} row) and the best solution found for the remaining instances (2^{nd} row) for makespan (1^{st} column) and overtime (2^{nd} column) minimization. On the 2^{nd} row, gray dots are instances solved by the CalendarOvertime model and black dots are for when all models timeout.

7.2 Comparing the CalendarOvertime and CumulativeOvertime Models

Figure 3 shows that the CumulativeOvertime model has an average speedup of 1.14 over the CalendarOvertime model for makespan optimization and 1.24 for overtime optimization, respectively, on the 1712 and 1436 instances solved optimally by both models. When comparing the best solutions found on the remaining instances, we see that, for makespan minimization, the CalendarOvertime model never proves optimality or finds a better solution than the CumulativeOvertime model. The CumulativeOvertime model proves optimality on respectively 1.6% and 1.4% of both these 423 makespan instances and the 699 overtime instances. It finds better solutions in 31% (44%) of makespan (overtime) instances. However, here, there are 5 overtime instances for which the CalendarOvertime model finds a better solution, and 1 where it proves optimality.

Thus, the CalendarOvertime constraint is a notable enhancement over the decomposition and is further improved by the CumulativeOvertime constraint.

8 Conclusion

We propose two new constraints to solve the cumulative scheduling problem with calendars and overtime. The CalendarOvertime constraint uses a precomputed substructure to enforce bounds consistency on the S_i, E_i, and O_i variables in $\mathcal{O}(|\text{dom}(S_i)|)$. The CumulativeOvertime constraint adapts the Time-Tabling rule to take calendars into account. Experiments on PSPLIB, BL, and PACK instances augmented with calendars show that the models using the specialized propagators of the new constraints outperform a model using a decomposition, the CumulativeOvertime constraints being a further enhancement over the CalendarOvertime constraints. These new constraints could also help solve the resource investment problem, the multi-mode resource-constraint project scheduling problem or even disjunctive problems such as job shop when they are augmented with calendars and overtime.

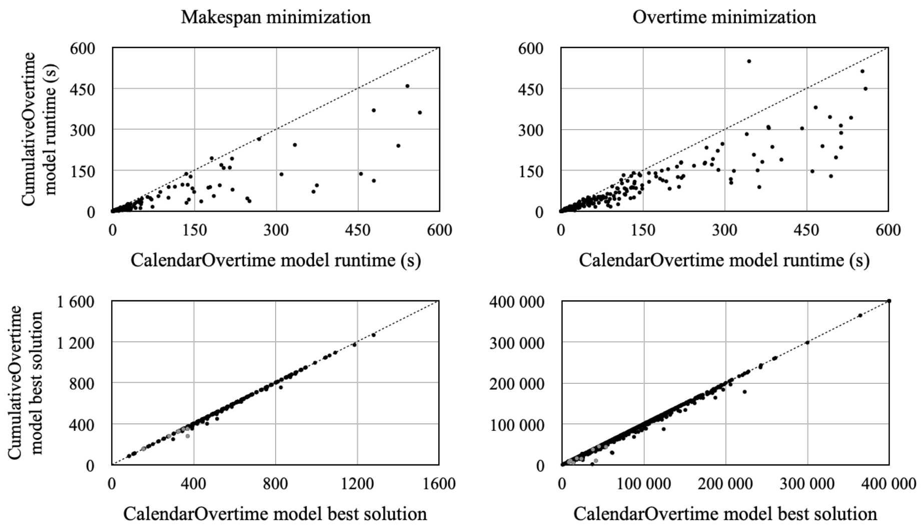


Figure 3 Comparison, between the CalendarOvertime and CumulativeOvertime models, of the runtime on the instances solved by both models (1st row) and the best solution found for the remaining instances (2nd row) for makespan (1st column) and overtime (2nd column) minimization. On the 2nd row, gray dots are instances solved by the CalendarOvertime model and black dots are for when all models timeout.

References

1 Abderrahmane Aggoun and Nicolas Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57–73, 1993. doi:10.1016/0895-7177(93)90068-A.

2 Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems.* Springer, 2001.

3 Philippe Baptiste and Claude Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, 5(1):119–139, 2000. doi:10.1023/A:1009822502231.

4 Nicolas Beldiceanu. Global constraints as graph properties on structured network of elementary constraints of the same type. *SICS Technical report T2000-01*, 2000.

5 Global constraint catalog: Calendar. https://sofdem.github.io/gccat/gccat/Ccalendar.html#uid15664. Accessed: 2024-04-04.

6 Raphaël Boudreault, Vanessa Simard, Daniel Lafond, and Claude-Guy Quimper. A constraint programming approach to ship refit project scheduling. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, volume 235, pages 10:1–10:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CP.2022.10.

7 Jacques Carlier and Emmanuel Néron. On linear lower bounds for the resource constrained project scheduling problem. *European Journal of Operational Research*, 149(2):314–324, 2003. doi:10.1016/S0377-2217(02)00763-4.

8 Geoffrey Chu. *Improving Combinatorial Optimization.* PhD thesis, The University of Melbourne, 2011.

9 Michael R. Garey and David S. Johnson. *Computers and intractability : a guide to the theory of NP-Completeness.* A Series of books in the mathematical sciences. W.H. Freeman, 1979.

10 Steven Gay, Renaud Hartert, and Pierre Schaus. Simple and scalable time-table filtering for the cumulative constraint. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP 2015)*, pages 149–157. Springer International Publishing, 2015. doi:10.1007/978-3-319-23219-5_11.

11 Rainer Kolisch and Arno Sprecher. Psplib - a project scheduling problem library: Or software - orsep operations research software exchange program. *European Journal of Operational Research*, 96(1):205–216, 1997. doi:10.1016/S0377-2217(96)00170-1.

12 Stefan Kreter, Julia Rieck, and Jürgen Zimmermann. Models and solution procedures for the resource-constrained project scheduling problem with general temporal constraints and calendars. *European Journal of Operational Research*, 251(2):387–403, 2016. doi:10.1016/j.ejor.2015.11.021.

13 Stefan Kreter, Andreas Schutt, and Peter J Stuckey. Using constraint programming for solving RCPSP/max-cal. *Constraints*, 22(3):432–462, 2017. doi:10.1007/s10601-016-9266-6.

14 Stefan Kreter, Andreas Schutt, Peter J. Stuckey, and Jürgen Zimmermann. Mixed-integer linear programming and constraint programming formulations for solving resource availability cost problems. *European Journal of Operational Research*, 266(2):472–486, 2018. doi:10.1016/j.ejor.2017.10.014.

15 Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP optimizer for scheduling. *Constraints*, 23(2):210–250, 2018. doi:10.1007/s10601-018-9281-x.

16 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, pages 529–543. Springer Berlin Heidelberg, 2007. doi:10.1007/978-3-540-74970-7_38.

17 Wilhelmus Petronella Maria Nuijten. *Time and resource constrained scheduling: a constraint satisfaction approach*. PhD thesis, Technische Universiteit Eindhoven, 1994.

18 Olga Ohrimenko, Peter J Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009. doi:10.1007/s10601-008-9064-x.

19 Pierre Ouellet and Claude-Guy Quimper. Time-table extended-edge-finding for the cumulative constraint. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP 2013)*, pages 562–577. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-40627-0_42.

20 Andreas Schutt, Thibaut Feydy, Peter J Stuckey, and Mark G Wallace. Explaining the cumulative propagator. *Constraints*, 16(3):250–282, 2011. doi:10.1007/s10601-010-9103-2.