# Synchronized Sweep Algorithms for Scalable Scheduling Constraints

Arnaud Letort
TASC team (CNRS/INRIA), Mines de Nantes, FR-44307 Nantes, France
Arnaud.Letort@mines-nantes.fr

Mats Carlsson
SICS, P.O. Box 1263, SE-164 29 Kista, Sweden
Mats.Carlsson@sics.se

Nicolas Beldiceanu
TASC team (CNRS/INRIA), Mines de Nantes, FR-44307 Nantes, France
Nicolas.Beldiceanu@mines-nantes.fr

SICS Technical Report T2013:05
ISSN: 1100-3154
ISRN: SICS-T–2013/05-SE


**Abstract:** This paper introduces a family of synchronized sweep based filtering algorithms for handling scheduling problems involving resource and precedence constraints. The key idea is to filter all constraints of a scheduling problem in a synchronized way in order to scale better. In addition to normal filtering mode, the algorithms can run in *greedy* mode, in which case they perform a greedy assignment of start and end times. The filtering mode achieves a significant speed-up over the decomposition into independent *cumulative* and *precedence* constraints, while the greedy mode can handle up to 1 million tasks with 64 resources constraints and 2 million precedences. These algorithms were implemented in both CHOCO and SICStus.

**Keywords:** Global Constraint; Sweep; Scheduling; Filtering Algorithm.

April 11, 2013

# Contents

# 1 Introduction

In the 2011 Panel of the Future of CP [1], one of the identified challenges for CP was the need to handle large scale problems. Multi-dimensional bin-packing problems were quoted as a typical example [2], particularly relevant in the context of cloud computing. Indeed, the importance of multi-dimensional bin-packing problems was recently highlighted in [3], and was part of the topic of the 2012 Roadef Challenge [4].

Till now, the tendency has been to use dedicated algorithms and metaheuristics [5] to cope with large instances. Various reasoning methods can be used for *cumulative* constraints, including Decomposition [6], Time-Table [7], Edge-Finding [8, 9], Energetic Reasoning [10], and recently Time-Table and Edge-Finding combined [11]. A comparison between these methods can be found in [10]. These filtering algorithms focus on having the best possible deductions rather than on scalability issues. This explains why they usually focus on small size problems (i.e., typically less than 200 tasks up to 10000 tasks) but leave open the scalability issue.

Like what was already done for the *geost* constraint [12], which handles up to 2 million boxes, our goal is to come up with lean filtering algorithms for cumulative problems. In order to scale well in terms of memory, we design lean filtering algorithms, which can also be turned into greedy algorithms. This approach allows us to avoid the traditional memory bottleneck problem of CP solvers due to trailing or copying data structures [13]. Moreover, like for *geost*, our lean algorithms and their derived greedy modes are compatible in the sense that they can be used both at the root and at each node of the search tree, i.e. first call the greedy mode for trying to find a solution and, if that doesn't work, use the filtering mode to restrict the variables and continue the search.

To achieve scalability we reuse the idea of *sweep synchronization* introduced in [14]: rather than propagating each constraint independently, we adjust the minimum (respectively maximum) of each variable wrt. all *cumulative* and all *precedence* constraints in one single sweep over the time horizon.

This report focuses on the *cumulative* constraint, originally introduced in [15] for modeling resource scheduling problems, and two extensions: (1) the *k-dimensional cumulative* constraint, which handles multiple parallel resources; and (2) the *k-dimensional cumulative with precedences* constraint, which handles multiple parallel resources and precedence relations between the tasks.

Given $n$ tasks and a resource with a maximum capacity $limit$ where each task $t$ ($0 \leq t < n$) is described by its start $s_t$, fixed duration $d_t$ ($d_t > 0$), end $e_t$ and fixed resource consumption $h_t$ ($h_t \geq 0$), the *cumulative* constraint with the two arguments

- $\langle\langle s_0, d_0, e_0, h_0\rangle, \ldots, \langle s_{n-1}, d_{n-1}, e_{n-1}, h_{n-1}\rangle\rangle$
- $limit$

holds if and only if conditions (1) and (2) are true:

$$\forall t \in [0, n-1] \ : s_t + d_t = e_t \tag{1}$$

$$\forall i \in \mathbb{Z} \ : \sum_{\substack{t \in [0,n-1], \\ i \in [s_t, e_t)}} h_t \leq limit \tag{2}$$

Section 2 provides a critical analysis of the major bottlenecks of the 2001 sweep algorithm [16] for *cumulative*, and gives general design decisions to avoid them. Based on these design decisions Section 3 presents a new sweep based filtering algorithm introduced in [17]. Section 4 revisits the sweep based filtering algorithm proposed in [18] for handling $k$ cumulative constraints, while Section 5 extends it to also handle precedence constraints between tasks. Section 6 puts these algorithms into perspective. Section 7 evaluates our new sweep algorithms on industrial and random instances, and on the well known PSPLib [19], and finally Section 8 concludes.

# 2 Motivations and General Decisions

The *sweep* algorithm is based on an idea which is widely used in computational geometry and that is called sweep [20]. In constraint programming, sweep was used for implementing the *non-overlapping* constraint [12] as well as the *cumulative* constraint [16].

In 2 dimensions, a plane *sweep* algorithm solves a problem by moving a vertical line, i.e. the *sweep-line*, from left to right. The algorithm uses two data structures:

- The *sweep-line status*, which contains some information related to the current position $\delta$ of the vertical line.
- The *event point series*, which holds the events to process, ordered in increasing order according to the time axis.

The algorithm initializes the sweep-line status for the starting position of the vertical line. Then the sweep-line "jumps" from event to event; each event is handled and inserted into or removed from the sweep-line status.

We recall the 2001 sweep algorithm [16] and identify five weaknesses preventing it from scaling well. To overcome those weaknesses, we introduce some general design decisions that will be shared by all the three new sweep algorithms described in Sections 3, 4 and 5.

## 2.1 A Critical Analysis of the 2001 Sweep Algorithm

In the context of resource scheduling, the sweep-line scans the time axis in order to build a compulsory part profile (CPP) and to perform checks and pruning according to this profile and to the resource limit. So the algorithm is a sweep variant of the *timetable* method [21]. To define the notion of CPP let us first introduce the definition of the *compulsory part* of a task.

**Definition 1** *(Compulsory Part) The compulsory part of a task $t$ is the intersection of all its feasible instances. The height of the compulsory part of a task $t$ at a given time point $i$ is defined by $h_t$ if $i \in [\overline{s_t}, \underline{e_t})$ and 0 otherwise, where $\overline{s_t}$ and $\underline{e_t}$ respectively denote the maximum value of the start variable $s_t$ and the minimum value of the end variable $e_t$.*

**Definition 2** *(CPP) Given a set of tasks $\mathcal{T}$, the CPP of the set $\mathcal{T}$ consists of the aggregation of the compulsory parts of the tasks in $\mathcal{T}$. The height of the CPP at a given instant $i$ is given by $\sum_{\substack{t \in \mathcal{T}, \\ i \in [\overline{s_t}, \underline{e_t})}} h_t$.*

We now introduce a running example that will be used, and extended later, throughout this report for illustrating the different algorithms.

**Example 1** *Consider five tasks $t_0$,$t_1$,...,$t_4$ which have the following* start, duration, end *and* height*:*

- $t_0$: $s_0 \in [1,1]$, $d_0 = 1$, $e_0 \in [2,2]$, $h_0 = 2$,
- $t_1$: $s_1 \in [0,3]$, $d_1 = 2$, $e_1 \in [2,5]$, $h_1 = 2$,
- $t_2$: $s_2 \in [0,5]$, $d_2 = 2$, $e_2 \in [2,7]$, $h_2 = 1$,
- $t_3$: $s_3 \in [0,9]$, $d_3 = 1$, $e_3 \in [1,10]$, $h_3 = 1$,
- $t_4$: $s_4 \in [0,7]$, $d_4 = 3$, $e_4 \in [3,10]$, $h_4 = 2$,

*subject to the constraint*

$$\begin{array}{ll} cumulative(\langle & \langle s_0, d_0, e_0, h_0 \rangle, \\ & \langle s_1, d_1, e_1, h_1 \rangle, \\ & \langle s_2, d_2, e_2, h_2 \rangle, \\ & \langle s_3, d_3, e_3, h_3 \rangle, \\ & \langle s_4, d_4, e_4, h_4 \rangle \rangle, 3) \end{array}$$

*(see Part (A) of Figure 1). Since task $t_0$ starts at instant 1 and since $t_1$ cannot overlap $t_0$ without exceeding the resource limit 3, the earliest start of task $t_1$ is adjusted to 2 (see Part (B)). Since task $t_1$ now has a compulsory part on interval $[3, 4)$ and since task $t_4$ cannot overlap that compulsory part without exceeding the resource limit 3, the earliest start of task $t_4$ is adjusted to 4 (see Part (C)). The purpose of the sweep algorithm is to perform such filtering in an efficient way.* □

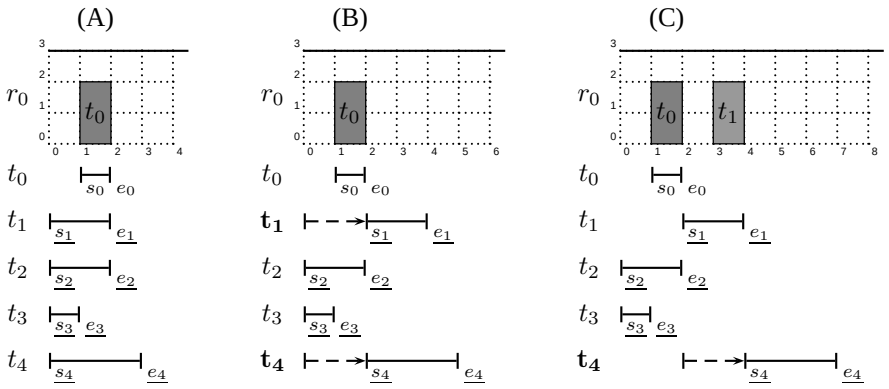


Figure 1: Parts (A), (B), and (C) respectively represent the earliest positions of the tasks and the CPP, of the initial problem described in Example 1, after a first sweep, and after a second sweep.

### 2.1.1 Event Point Series

In order to build the CPP and to prune the start variables of the tasks, the sweep algorithm considers the following types of events:

- *Profile events* for building the CPP correspond to the latest starts and the earliest ends of the tasks for which the latest start is strictly less than the earliest end (i.e. the start and the end of a non-empty compulsory part).
- *Pruning events* for recording the tasks to prune, i.e. the not yet fixed tasks that intersect the current position $\delta$ of the sweep-line.

Table 1 describes the different types of events, where each event corresponds to a quadruple $\langle event\ type, task\ generating\ the\ event, event\ date, available\ space\ update \rangle$. These events are sorted by increasing date.

Table 1: Event types for the 2001 sweep with corresponding condition for generating them. The last event attribute is only relevant for event types $SCP$ and $ECP$.

| **Generated Events (2001 algo.)** | **Conditions** |
|---|---|
| $\langle SCP, t, \overline{s_t}, -h_t \rangle$ | $\overline{s_t} < \underline{e_t}$ |
| $\langle ECP, t, \underline{e_t}, +h_t \rangle$ | $\overline{s_t} < \underline{e_t}$ |
| $\langle PR, t, \underline{s_t}, 0 \rangle$ | $\underline{s_t} \neq \overline{s_t}$ |

*Continuation of Example 1 (Generated Events).* To the initial domains of the five tasks of Example 1 correspond the following events that are sorted by increasing dates: $\langle PR, 1, 0, 0 \rangle$ $\langle PR, 2, 0, 0 \rangle$ $\langle PR, 3, 0, 0 \rangle$ $\langle PR, 4, 0, 0 \rangle$ $\langle SCP, 0, 1, -2 \rangle$ $\langle ECP, 0, 2, 2 \rangle$. □

### 2.1.2 Sweep-Line Status

The sweep-line maintains three pieces of information:

- The current sweep-line position $\delta$, initially set to the date of the first event.
- The amount of available resource at instant $\delta$, denoted by $gap$, i.e., the difference between the resource limit and the height of the CPP at instant $\delta$.
- A list of tasks $\mathcal{T}_{prune}$, recording all tasks that potentially can overlap $\delta$, i.e. tasks for which the start may be pruned wrt. instant $\delta$.

The sweep algorithm first creates and sorts the events wrt. their dates. Then, the sweep-line moves from one event to the next event, updating $gap$ and $\mathcal{T}_{prune}$. Once all events at $\delta$ have been handled, the sweep algorithm tries to prune all tasks in $\mathcal{T}_{prune}$ wrt. $gap$ and interval $[\delta, \delta_{next})$ where $\delta_{next}$ is the next sweep-line position, i.e. the date of the next event. More precisely, given a task $t \in \mathcal{T}_{prune}$ with no compulsory part overlapping interval $[\delta, \delta_{next})$ such that $h_t > gap$, the interval $[\delta - d_t + 1, \delta_{next})$ is removed from the start variable of task $t$.

*Continuation of Example 1 (Illustrating the 2001 Sweep Algorithm).* The sweep algorithm reads the events $\langle PR, 1, 0, 0\rangle$, $\langle PR, 2, 0, 0\rangle$, $\langle PR, 3, 0, 0\rangle$, $\langle PR, 4, 0, 0\rangle$ and sets $gap$ to the resource limit 3 and $\mathcal{T}_{prune}$ to $\{t_1, t_2, t_3, t_4\}$. During a first sweep, the compulsory part of task $t_0$ (see Part (A) of Figure 1) permits to prune the start of $t_1$ and $t_4$ since the $gap$ on $[1, 2)$ is strictly less than $h_1$ and $h_4$. The pruning of the earliest start of $t_1$ during the first sweep causes the creation of a compulsory part for task $t_1$ which is not immediately used to perform more pruning (see Part (B) of Figure 1). As shown in Part (C) of Figure 1, it is necessary to wait for a second sweep to take advantage of this new compulsory part to adjust the earliest start of task $t_4$ to 4. A third and last sweep is performed to find out that the fixpoint was reached. □

### 2.1.3 Weaknesses of the 2001 Sweep Algorithm

We now list the main weaknesses of the 2001 sweep algorithm.

① `[Too static]` The potential increase of the CPP during a single sweep is not dynamically taken into account. In other words, creations and extensions of compulsory parts during a sweep are not immediately used to perform more pruning while sweeping. Example 1 illustrates this point since the 2001 sweep algorithm needs to be run from scratch 3 times before reaching its fixpoint.

② `[Often reaches its worst-case time complexity]` The worst-case time complexity of the 2001 sweep algorithm is $O(n^2)$ where $n$ is the number of tasks. This complexity is often reached in practice when most of the tasks can be placed everywhere on the time line. The reason is that it needs at each position $\delta$ of the sweep-line to systematically re-scan all tasks that overlap $\delta$. Profiling the 2001 implementation indicates that the sweep algorithm spends up to 45% of its overall running time scanning again and again the list of potential tasks to prune.

③ `[Creates holes in the domains]` The 2001 sweep algorithm removes intervals of consecutive values from domain variables. This is a weak point, which prevents handling large instances since the domain of a variable cannot just be compactly represented by its minimum and maximum values.

④ `[Does not take advantage of bin-packing]` For instances where all tasks have duration one, the worst time complexity $O(n^2)$ is left unchanged.

⑤ `[Too local]` Having in the same problem multiple *cumulative* constraints that systematically share variables leads to the following source of inefficiency. In a traditional setting, each *cumulative* constraint is propagated independently on all its variables, and because of the shared variables, the sweep algorithm of each *cumulative* constraint should be rerun several times to reach the fixpoint. Note that a single update of a bound of a variable by one *cumulative* constraint will trigger all the other *cumulative* constraints again. The same observation holds when, in addition to resource constraints, one also considers precedences between tasks.

## 2.2 General Design Decisions

We now give some important general design decisions that permit to avoid the five weaknesses of the 2001 sweep algorithm identified above. Then, we introduce the property maintained by our sweep algorithm [17] for one single *cumulative* constraint, which will be extended in Sections 4 and 5 for the *k-dimensional cumulative* and the *k-dimensional cumulative with precedences* constraints.

### 2.2.1 Handling the Weaknesses of the 2001 sweep

**Avoiding Point ①** `[Too static]`. As illustrated by Example 1, the 2001 sweep algorithm needs to be re-run several times in order to reach its fixpoint (i.e., 3 times in our example). This is due to the fact that, during one sweep, restrictions on task origins are not immediately taken into account. The three new sweep algorithms filter the task origins in two distinct sweep stages. A first stage, called *sweep_min*, tries to adjust the earliest starts of tasks by performing a sweep from left to right, and a second stage, called *sweep_max*, tries to adjust the latest ends by performing a sweep from right to left. Note that the propagator needs to iterate the two phases until fixpoint. Suppose that *sweep_min* has run, and that *sweep_max* extends the CPP. Then *sweep_min* may no longer be at fixpoint, and needs to run again, and so on. W.l.o.g, we focus from now on the first stage, *sweep_min*, since the second stage is completely symmetric. In our three new algorithms, *sweep_min* dynamically uses these deductions to reach its fixpoint in one single sweep. To deal with this aspect, our new sweep algorithms introduce the concept of *conditional events*, i.e., events that are created while sweeping over the time axis, and *dynamic events*, i.e., events that can be shifted over the time axis.

**Avoiding Point ②** `[Often reaches its worst-case time complexity]`. For partially avoiding Point ② due to the rescan of all tasks that overlap the current sweep-line position, we introduce dedicated data structures in our three new algorithms. The idea is based on the following observations: if a task of height $h$ cannot overlap the current sweep-line position and consequently needs to be adjusted, all tasks with a height greater than or equal to $h$ need to be adjusted too; and symmetrically, if a task of height $h$ can overlap the current sweep-line position, all tasks with a height less than or equal to $h$ can also overlap the current sweep-line position and consequently do not need to be adjusted too.

**Avoiding Point ③** `[Creates holes in the domains]`. The first difference from the 2001 sweep is that our algorithms only deal with domain bounds, which is a good way to reduce the memory consumption for the representation of domain variables. Consequently, we need to change the 2001 algorithm, which creates holes in the domain of task origins.

**Avoiding Point ④** `[Does not take advantage of bin-packing]`. Moreover, the data structures introduced for avoiding Point ② will permit to reduce the worst-case time complexity of our algorithms in the specific case of bin-packing problems, i.e. when the duration of all tasks is reduced to one. This point will be explained in Section 3.5.

**Avoiding Point ⑤** `[Too local]`. To handle this weak point, we first design a second filtering algorithm that handles multiple parallel resources in one single constraint, called *k-dimensional cumulative*. The main difference is that we directly adjust the earliest start of a task wrt. all resource constraints rather than successively and completely propagating each resource constraint independently. Second, following this idea, we also design a third filtering algorithm that handles multiple parallel resources and precedences in one single constraint, called *k-dimensional cumulative with precedences*. First, we recall a method for adjusting the start and end times of a set of tasks subject to a set of precedences. Then, we present the main idea of these two filtering algorithms.

**Handling a Set of Precedences.** Given a set of tasks $\mathcal{T}$ and a set of precedences where each precedence denotes a task of $\mathcal{T}$ that must be completed before the start of another task of $\mathcal{T}$, adjusting the earliest and

latest start of each task is done by a two-phase algorithm that starts from a topological order of the tasks (each task is a vertex of a digraph and each precedence an arc):

① The first phase adjusts the earliest start of each task by successively selecting a source, i.e. a task with no predecessor, removing it and updating the earliest start of its direct successors.

② Similarly, the second phase adjusts the latest start of each task by successively selecting a sink, i.e. a task with no successor.

Since in each phase the method considers each task only once, it converges directly to the fixpoint in linear time. The key observation is that the adjustment of the earliest start of a task does not influence the earliest start of its predecessors.

**Importing the Idea of Topological Sort.** As soon as resource constraints come into play, the two-phase method for handling a set of precedences was not considered any more and each resource and precedence constraints were propagated independently until the fixpoint. The key idea of this report is to reuse as much as possible the idea of the two-phase method by selecting, in the first phase, the task which has the earliest start and adjusting its earliest start *wrt. all constraints* where the task is involved, i.e. all precedence and resource constraints. To achieve this, we revisit the way resource and precedence constraints are propagated so that we consider them in a synchronized way rather than in isolation.

### 2.2.2 Property

Our dynamic sweep algorithm for the *cumulative* constraint maintains the following property.

**Property 1** *Given a* cumulative *constraint with its set of tasks $\mathcal{T}$ and resource limit $limit$,* sweep_min *ensures that:*

$$\forall t \in \mathcal{T}, \forall i \in [\underline{s_t}, \underline{e_t}) \ : \ h_t + \sum_{\substack{t' \in \mathcal{T} \setminus \{t\}, \\ i \in [\overline{s_{t'}}, \underline{e_{t'}})}} h_{t'} \leq limit \tag{3}$$

Property 1 ensures that, for any task $t$ of the *cumulative* constraint, one can schedule $t$ at its earliest start without exceeding the resource limit wrt. the CPP for the tasks of $\mathcal{T} \setminus \{t\}$.

Note that we can construct from Property 1 a relaxed solution of the *cumulative* constraint by:

① setting the resource consumption to 0 for the tasks that do not have any compulsory part,

② setting the duration to the size of the compulsory part (i.e. $\underline{e_t} - \overline{s_t}$) for the tasks that do have a compulsory part, and

③ assigning the start of each task to its earliest start.

# 3 A Dynamic Sweep Algorithm for one Single *cumulative* Constraint

This section presents the new sweep algorithm introduced in [17] for the *cumulative* constraint. We describe it in a similar way the 2001 original sweep algorithm was introduced in Section 2. We first present the new *event point series*, then describe the new *sweep-line status*, and the overall algorithm. Finally we prove that Property 1 introduced above is maintained by the new algorithm and we give its worst-case complexity in the general case as well as in the case where all task durations are fixed to one.

## 3.1 Event Point Series

In order to address Point ① `[Too static]` of Section 2, *sweep_min* should handle the extension and the creation of compulsory parts caused by the adjustment of the earliest starts of tasks in one single sweep. We therefore need to modify the events introduced in Table 1. Table 2 presents the events of *sweep_min* and their relations with the events of the 2001 algorithm.

- The event type $\langle SCP, t, \overline{s_t}, -h_t \rangle$ for the start of compulsory part of task $t$ is left unchanged. Note that, since *sweep_min* only adjusts the earliest starts, the start of a compulsory part (which corresponds to a latest start) can never be further extended to the left.
- The event type $\langle ECPD, t, \underline{e_t}, h_t \rangle$ for the end of the compulsory part of task $t$ is converted to $\langle ECPD, t, \underline{e_t}, h_t \rangle$ where $D$ stands for *dynamic*. The date of such event corresponds to the earliest end of $t$ (also the end of its compulsory part) and may increase due to the adjustment of the earliest start of task $t$.
- A new event type $\langle CCP, t, \overline{s_t}, 0 \rangle$, where $CCP$ stands for *conditional compulsory part*, is initially created for each task $t$ that does not have any compulsory part. At the latest, once the sweep-line reaches position $\overline{s_t}$, it adjusts the earliest start of task $t$ to know if a compulsory part appears. Consequently the conditional event can be transformed into an $SCP$ and an $ECPD$ events, reflecting the creation of compulsory part for a task that did not initially have any compulsory part.
- The event type $\langle PR, t, \underline{s_t}, 0 \rangle$ for the earliest start of task $t$ is left unchanged. It is required to add task $t$ to the list of tasks that potentially can overlap $\delta$.

Table 2: The list of different event types with the condition for generating them. The last attribute of an event (i.e. *available space increment*) is only relevant for $SCP$, $ECP$ and $ECPD$ event types.

| **New Events** | **Events (2001 algo.)** | **Conditions** |
|---|---|---|
| $\langle SCP, t, \overline{s_t}, -h_t \rangle$ | $\langle SCP, t, \overline{s_t}, -h_t \rangle$ | $\overline{s_t} < \underline{e_t}$ |
| $\langle ECPD, t, \underline{e_t}, +h_t \rangle$ | $\langle ECP, t, \underline{e_t}, +h_t \rangle$ | $\overline{s_t} < \underline{e_t}$ |
| $\langle CCP, t, \overline{s_t}, 0 \rangle$ | | $\overline{s_t} \geq \underline{e_t}$ |
| $\langle PR, t, \underline{s_t}, 0 \rangle$ | $\langle PR, t, \underline{s_t}, 0 \rangle$ | $\underline{s_t} \neq \overline{s_t}$ |

On the one hand, some of these events may have their dates modified while sweeping (see $ECPD$). On the other hand, some events create new events (see $CCP$). Consequently, rather than just sorting all events initially, we insert them by increasing date into a heap called $h\_events$ so that new or updated events can be dynamically added into this heap while sweeping.

*Continuation of Example 1 (New Generated Events for* sweep_min*).* The following events are generated and added into $h\_events$ (note that the new events are highlighted in bold): $\langle PR, 1, 0, 0 \rangle$, $\langle PR, 2, 0, 0 \rangle$, $\langle PR, 3, 0, 0 \rangle$, $\langle PR, 4, 0, 0 \rangle$, $\langle SCP, 0, 1, -2 \rangle$, $\langle \mathbf{ECPD}, \mathbf{0}, \mathbf{2}, \mathbf{2} \rangle$, $\langle \mathbf{CCP}, \mathbf{1}, \mathbf{3}, \mathbf{0} \rangle$, $\langle \mathbf{CCP}, \mathbf{2}, \mathbf{5}, \mathbf{0} \rangle$, $\langle \mathbf{CCP}, \mathbf{4}, \mathbf{7}, \mathbf{0} \rangle$, $\langle \mathbf{CCP}, \mathbf{3}, \mathbf{9}, \mathbf{0} \rangle$. The event $\langle \mathbf{ECPD}, \mathbf{0}, \mathbf{2}, \mathbf{2} \rangle$ stands for the end of compulsory part of task $t_0$. In our example, since task $t_0$ is fixed, this event cannot be pushed on the time axis. The event $\langle \mathbf{CCP}, \mathbf{1}, \mathbf{3}, \mathbf{0} \rangle$ stands for the date where the compulsory part of task $t_1$ can start if and only if its earliest start is pruned enough (i.e. such that $\underline{s_t} + d_t > \overline{s_t}$). □

## 3.2 Sweep-Line Status

The sweep-line maintains the following pieces of information:

- The current sweep-line position $\delta$, initially set to the date of the first event.
- The amount of available resource at instant $\delta$, denoted by $gap$, i.e., the difference between the resource limit and the height of the CPP.
- Two heaps $h\_conflict$ and $h\_check$ for partially avoiding Point ② of Section 2, namely avoiding to scan again and again the tasks that overlap the current sweep-line position. W.l.o.g. assume that the sweep-line is at its initial position and that we handle an event of type $PR$ (i.e., we try to find out the earliest possible start of a task $t$).

– On the one hand, if the height of task $t$ is strictly greater than the available gap at $\delta$, we know that we have to adjust the earliest start of task $t$. In order to avoid re-checking each time we move the sweep-line, whether or not the gap is big enough wrt. $h_t$, we say that task $t$ is in *conflict* with $\delta$. We insert task $t$ into the heap $h\_conflict$, which records all tasks that are in conflict with $\delta$, sorted by increasing height, i.e. the top of the heap $h\_conflict$ corresponds to the smallest value. This ordering is induced by the fact that, if we need to adjust the earliest start of a task $t$, all earliest task starts with a height greater than or equal to $h_t$ also need to be adjusted.

– On the other hand if the height of task $t$ is less than or equal to the available gap at instant $\delta$, we know that the earliest start of task $t$ could be equal to $\delta$. But to be sure, we need to check Property 1 for task $t$ (i.e., $\mathcal{T} = \{t\}$). For this purpose we insert task $t$ into the heap $h\_check$, which records all tasks for which we currently check Property 1. Task $t$ stays in $h\_check$ until a conflict is detected (i.e., $h_t$ is greater than the available gap, and $t$ goes back into $h\_conflict$) or until the sweep-line passes instant $\delta + d_t$ without having detected any conflict (and we have found a feasible earliest start of task $t$ wrt. Property 1). In the heap $h\_check$, tasks are sorted by decreasing height, i.e. the top of the heap $h\_check$ corresponds to the largest value, since if a task $t$ is not in conflict with $\delta$, all other tasks of $h\_check$ of height less than or equal to $h_t$ are also not in conflict with $\delta$.
In the following algorithms, function $\mathrm{empty}(h)$ returns **true** if the heap $h$ is empty, **false** otherwise. Function $\mathrm{get\_top\_key}(h)$ returns the key of the top element in the heap $h$. We introduce an integer array *mins*, which stores for each task $t$ in $h\_check$ the value of $\delta$ when $t$ was added into $h\_check$.

## 3.3 Algorithm

The *sweep_min* algorithm performs one single sweep over the event point series in order to adjust the earliest start of the tasks wrt Property 1. It consists of a main loop, a filtering part and a synchronization part. This last part is required in order to directly handle the deductions attached to the creation or increase of compulsory parts in one single sweep. In addition to *mins* and the heaps $h\_check$ and $h\_conflict$, we introduce a Boolean array *evup*, which indicates for each task $t$ whether events related to the compulsory part of task $t$ were updated or not. The value is set to **true** once we have found the final value of the earliest start of task $t$ and once the events related to the compulsory part of task $t$, if it exists, are up to date in the heap of events. We introduce a list *newActiveTasks*, which records all tasks that have their $PR$ event at $\delta$. The primitive $\mathrm{adjust\_min\_var}(var, val)$ adjusts the minimum value of the variable $var$ to value $val$.

### 3.3.1 Main Loop

The main loop (Algorithm 1) consists of:

- `[INITIALIZATION]` (lines 2 to 6). The events are generated and inserted into $h\_events$ according to the conditions given in Table 2. The $h\_check$ and $h\_conflict$ heaps are initialized as empty heaps. The Boolean $evup_t$ is set to **true** if and only if the task $t$ is fixed. The integer $mins_t$ is set to the earliest start of the task $t$. The list *newActiveTasks* is initialized as an empty list. $\delta$ is set to the date of the first event.

- `[MAIN LOOP]` (lines 8 to 25). For each date $\delta$ the main loop processes all the corresponding events. It consists of the following parts:

  – `[HANDLING A SWEEP-LINE MOVE]` (lines 10 to 17). Each time the sweep-line moves, we update the sweep-line status ($h\_check$ and $h\_conflict$) wrt. the new *active tasks*, i.e. the tasks for which the earliest start is equal to $\delta$. All the new active tasks that are in conflict with $\delta$ in the CPP are added into $h\_conflict$ (line 13). For tasks that are not in conflict we check whether the sweep interval $[\delta, \delta_{next})$ is big enough wrt. their durations. Tasks for which the sweep interval is too small are added into $h\_check$ (line 14). Then to take into account the new

**ALGORITHM** sweep_min() : boolean
1: [INITIALIZATION]
2: $h\_events \leftarrow$ generation of events wrt. $n$, $\underline{s_t}$, $\overline{s_t}$, $d_t$, $\underline{e_t}$ and $h$ and Table 2.
3: $h\_check$, $h\_conflict \leftarrow \emptyset$; $newActiveTasks \leftarrow \emptyset$
4: **for** $t = 0$ **to** $n - 1$ **do**
5: $evup_t \leftarrow (\underline{s_t} = \overline{s_t})$; $mins_t \leftarrow \underline{s_t}$
6: $\delta \leftarrow$ get_top_key$(h_events)$; $\delta_{next} \leftarrow \delta$; $gap \leftarrow limit$
7: [MAIN LOOP]
8: **while** $\neg$empty$(h_events)$ **do**
9: [HANDLING A SWEEP-LINE MOVE]
10: **if** $\delta \neq \delta_{next}$ **then**
11: **while** $\neg$empty$(newActiveTasks)$ **do**
12: extract first task $t$ from $newActiveTasks$
13: **if** $h_t > gap$ **then** add $\langle h_t, t\rangle$ into $h\_conflict$
14: **else if** $d_t > \delta_{next} - \delta$ **then** {add $\langle h_t, t\rangle$ into $h\_check$; $mins_t \leftarrow \delta$;}
15: **else** $evup_t \leftarrow$ **true**
16: **if** $\neg$filter_min$(\delta, \delta_{next})$ **then return false**
17: $\delta \leftarrow \delta_{next}$
18: [HANDLING CURRENT EVENT]
19: $\delta \leftarrow$ synchronize$(\delta)$
20: extract $\langle type, t, \delta, dec\rangle$ from $h\_events$
21: **if** $type = SCP \vee type = ECPD$ **then** $gap \leftarrow gap + dec$
22: **else if** $type = PR$ **then** $newActiveTasks \leftarrow newActiveTasks \cup \{t\}$
23: [GETTING NEXT EVENT]
24: **if** empty$(h_events) \wedge \neg$filter_min$(\delta, +\infty)$ **then return false**
25: $\delta_{next} \leftarrow$ synchronize$(\delta)$
26: **return true**

**Algorithm 1:** Returns **false** if a resource overflow is detected while sweeping, **true** otherwise. If **true**, ensures that the earliest start of each task is pruned so that Property 1 holds.

available space (i.e., $gap$) on top of the CPP, filter_min (see Algorithm 2) is called to update $h\_check$ and $h\_conflict$ and to adjust the earliest start of tasks for which a feasible position wrt. Property 1 was found.

– [HANDLING CURRENT EVENT] (lines 19 to 22). First, algorithm synchronize (line 19) (1) converts conditional events ($CCP$) to $SCP$ and $ECPD$ events, or ignore them if the corresponding task has no compulsory part, (2) pushes dynamic events ($ECPD$) to their right position to ensure events are sorted on their dates. Second, the top event is extracted from the heap $h\_events$. Depending of its type (i.e., $SCP$ or $ECPD$), the available resource is updated, or (i.e., $PR$), the task associated with the current event is added into the list of new active tasks (line 22).

– [GETTING NEXT EVENT] (lines 24 to 25). If there is no more event in $h\_events$, filter_min is called in order to empty the heap $h\_check$, which may generate new compulsory part events.

### 3.3.2 The Filtering Part

Once all the events associated with the current date $\delta$ are handled, Algorithm 2 takes into account the new available space on top of the CPP. It processes tasks in $h\_check$ and $h\_conflict$ in order to adjust the earliest start of the tasks wrt. Property 1. The main parts of the algorithm are:

- [CHECK RESOURCE OVERFLOW] (line 2). If the available resource $gap$ is negative on the sweep interval $[\delta, \delta_{next})$, Algorithm 2 returns **false** for failure (i.e. the resource capacity $limit$ is exceeded).

**ALGORITHM** $\text{filter\_min}(\delta, \delta_{next}) : \text{boolean}$

```
1:  [CHECK RESOURCE OVERFLOW]
2:  if gap < 0 then return false
3:  [UPDATING TOP TASKS OF h_check WRT gap]
4:  while ¬empty(h_check) ∧ (empty(h_events) ∨ get_top_key(h_check) > gap) do
5:     extract ⟨h_t, t⟩ from h_check
6:     if δ ≥ s̄_t ∨ δ − mins_t ≥ d_t ∨ empty(h_events) then
7:        if ¬adjust_min_var(s_t, mins_t) ∨ ¬adjust_min_var(e_t, mins_t + d_t) then return false
8:        if ¬evup_t then {update events of the compulsory part of t; evup_t ← true;}
9:     else
10:       add ⟨h_t, t⟩ into h_conflict
11: [UPDATING TOP TASKS OF h_conflict WRT gap]
12: while ¬empty(h_conflict) ∧ get_top_key(h_conflict) ≤ gap do
13:    extract ⟨h_t, t⟩ from h_conflict
14:    if δ ≥ s̄_t then
15:       if ¬adjust_min_var(s_t, s̄_t) ∨ ¬adjust_min_var(e_t, ē_t) then return false
16:       if ¬evup_t then {update events of the compulsory part of t; evup_t ← true;}
17:    else
18:       if δ_next − δ ≥ d_t then
19:          if ¬adjust_min_var(s_t, δ) ∨ ¬adjust_min_var(e_t, δ + d_t) then return false
20:          if ¬evup_t then {update events of the compulsory part of t; evup_t ← true;}
21:       else
22:          add ⟨h_t, t⟩ into h_check; mins_t ← δ;
23: return true
```

**Algorithm 2:** Tries to adjust the earliest starts of the top tasks in $h\_check$ and $h\_conflict$ wrt. the current sweep interval $\mathcal{I} = [\delta, \delta_{next}]$ and the available resource $gap$ on top of the CPP on interval $\mathcal{I}$. Returns **false** if a resource overflow is detected, **true** otherwise.

- `[UPDATING TOP TASKS OF` $h\_check$ `WRT` $gap$`]` (lines 4 to 10). All tasks in $h\_check$ of height greater than the available resource $gap$ are extracted.
  - A first case is when task $t$ has been in $h\_check$ long enough (i.e. $\delta - mins_t \geq d_t$, line 6), meaning that the task is not in conflict on interval $[mins_t, \delta)$, whose size is greater than or equal to the duration $d_t$ of task $t$. Consequently, we adjust the earliest start of task $t$ to value $mins_t$. Remember that $mins_t$ corresponds to the latest sweep-line position where task $t$ was moved into $h\_check$.
  - A second case is when $\delta$ has passed the latest start of task $t$ (i.e. $\delta \geq \overline{s_t}$, line 6). That means task $t$ was not in conflict on interval $[mins_t, \delta)$ either, and we can adjust its earliest start to $mins_t$.
  - A third case is when there is no more event in the heap $h\_events$ (i.e. $\text{empty}(h\_events)$, line 6). It means that the height of the CPP is equal to zero and we need to empty $h\_check$.
  - Otherwise, since the height of task $t$ is greater than the available resource, the task is added into $h\_conflict$ (line 10).
- `[UPDATING TOP TASKS OF` $h\_conflict$ `WRT` $gap$`]` (lines 13 to 23). All tasks in $h\_conflict$ that are no longer in conflict at $\delta$ are extracted. If $\delta$ is not located before the latest start of task $t$, we know that task $t$ cannot be scheduled before its latest start. Otherwise, we compare the duration of task $t$ with the size of the current sweep interval $[\delta, \delta_{next}]$ and decide whether to adjust the earliest start of task $t$ or to add task $t$ into $h\_check$.

### 3.3.3 The Synchronization Part

In order to handle dynamic and conditional events, Algorithm 3 checks and possibly updates the top event of the heap $h\_events$ before any access to $h\_events$ by the main algorithm *sweep_min*. The main parts of

**ALGORITHM** synchronize($\delta$) : integer

```
1:  [UPDATING TOP EVENTS]
2:  repeat
3:     if empty(h_events) then return −∞
4:     sync ← true; ⟨date, t, type, dec⟩ ← consult top event of h_events;
5:     [PROCESSING DYNAMIC (ECPD) EVENT]
6:     if type = ECPD ∧ ¬evup_t then
7:        if t ∈ h_check then update event date to mins_t + d_t
8:        else update event date to s̄_t + d_t
9:        evup_t ← true; sync ← false;
10:    [PROCESSING CONDITIONAL (CCP) EVENT]
11:    else if type = CCP ∧ ¬evup_t ∧ date = δ then
12:       if t ∈ h_check ∧ mins_t + d_t > δ then
13:          add ⟨SCP, t, δ, −h_t⟩ and ⟨ECPD, t, mins_t + d_t, h_t⟩ into h_events
14:       else if t ∈ h_conflict then
15:          add ⟨SCP, t, δ, −h_t⟩ and ⟨ECPD, t, ē_t, h_t⟩ into h_events
16:       evup_t ← true; sync ← false;
17: until sync
18: return date
```

**Algorithm 3:** Checks that the event at the top of $h\_events$ is updated and returns the date of the next event or $-\infty$ if $h\_events$ is empty.

the algorithm are:

- `[UPDATING TOP EVENTS]` (lines 2 to 17). Dynamic and conditional events require us to check whether the next event to be extracted by Algorithm 1 needs to be updated or not. The repeat loop updates the next event if necessary until the top event is up to date.
- `[PROCESSING DYNAMIC EVENT]` (lines 6 to 9). An event of type $ECPD$ must be updated if the related task $t$ is in $h\_check$ or in $h\_conflict$. If task $t$ is in $h\_check$, it means that its earliest start can be adjusted to $mins_t$. Consequently, its $ECPD$ event is updated to the date $mins_t + d_t$ (line 7). If task $t$ is in $h\_conflict$, it means that task $t$ cannot start before its latest starting time $\overline{s_t}$. Consequently, its $ECPD$ event is pushed back to the date $\overline{s_t} + d_t$ (line 8).
- `[PROCESSING CONDITIONAL EVENT]` (lines 11 to 16). When the sweep-line reaches the position of a $CCP$ event for a task $t$, we need to know whether or not a compulsory part for $t$ is created. As $evup_t$ is set to **false**, we know that $t$ is either in $h\_check$ or in $h\_conflict$. If task $t$ is in $h\_check$, a compulsory part is created if and only if $mins_t + d_t > \delta$ (lines 12 to 13). If task $t$ is in $h\_conflict$ the task is fixed to its latest position and related events are added into $h\_events$ (line 15).

*Continuation of Example 1 (Illustrating the Dynamic Sweep Algorithm).* The sweep algorithm first initializes the current sweep-line position to 0, i.e. the first event date, and the $gap$ to 3, i.e. the resource limit. The algorithm reads the four $PR$ events related to the tasks $t_1$, $t_2$, $t_3$ and $t_4$. Since the heights of tasks $t_1$, $t_2$, $t_4$ are less than or equal to the gap and their durations are strictly greater than the size of the sweep interval, these tasks are added into $h\_check$ (Algorithm 1, line 14). Task $t_3$ is not added into $h\_check$ since its duration is equal to the size of the sweep interval (Algorithm 1, line 14), i.e. $t_3$ cannot be adjusted. Then, it moves the sweep-line to the position 1, reads the event $\langle SCP, 0, 1, -2\rangle$ and sets $gap$ to 1. The call of filter_min with $\delta = 1$, $\delta_{next} = 2$ and $gap = 1$ retrieves $t_1$ and $t_4$ from $h\_check$ and inserts them into $h\_conflict$ (Algorithm 2, line 10). Then it moves the sweep-line to the position 2, reads the event $\langle ECPD, 0, 2, 2\rangle$ and sets $gap$ to 3. The call of filter_min with $\delta = 2$, $\delta_{next} = 3$ and $gap = 3$ retrieves $t_1$ and $t_4$ from $h\_conflict$ and inserts them into $h\_check$ (Algorithm 2, line 22). Then it moves the sweep-line to the position 3 and reads the event $\langle CCP, 1, 3, 0\rangle$. Since task $t_1$ is in $h\_check$ and its potential earliest end is greater than $\delta$ (Algorithm 3, line 12), the $CCP$ event of $t_1$ is converted into $\langle SCP, 1, 3, -2\rangle$ and $\langle ECPD, 1, 4, +2\rangle$ standing for the creation of a compulsory part on interval $[3, 4)$. The sweep-line reads

the new $SCP$ event related to task $t_1$ and sets $gap$ to 1. The call of filter_min with $\delta = 3$, $\delta_{next} = 4$ and $gap = 1$ retrieves $t_4$ from $h\_check$ and inserts it into $h\_conflict$. Then it moves the sweep-line to the position 4, reads the event $\langle ECPD, 1, 4, +2\rangle$ and sets $gap$ to 3. Since there is no more compulsory part, the earliest start of $t_4$ is adjusted to 4 and the fixpoint of *sweep_min* is reached. Note that the creation of the compulsory part occurs after the sweep-line position, which is key to ensuring Property 1. □

## 3.4 Correctness and Property Achieved by *sweep_min*

We now prove that after the termination of *sweep_min*(Algorithm 1), Property 1 holds. For this purpose, we first introduce the following lemma.

**Lemma 1** *At any point of its execution,* sweep_min*(Algorithm 1) cannot generate a new compulsory part that is located before the current position $\delta$ of the sweep-line.*

**Proof 1** *Since the start of the compulsory part of a task $t$ corresponds to its latest start $\overline{s_t}$, which is indicated by its $CCP$ or $SCP$ event, and since* sweep_min *only prunes earliest starts, the compulsory part of task $t$ cannot start before the date associated to this event. Consequently, the latest value of $\delta$ to know whether the compulsory part of task $t$ is created is $\overline{s_t}$. This case is processed by Algorithm 3, lines 11 to 16.*

*The end of the compulsory part of a task $t$ corresponds to its earliest end $\underline{e_t}$ and is indicated by its $ECPD$ event. To handle its potential extension to the right, the earliest start of task $t$ must be adjusted to its final position before the sweep extracts its $ECPD$ event. This case is processed by Algorithm 3, lines 6 to 9.* □

**Proof 2 (of Property 1)** *Given a task $t$, let $\delta_t$ and $min_t$ respectively denote the position of the sweep-line when the earliest start of task $t$ is adjusted by* sweep_min*, and the new earliest start of task $t$. We successively show the following points:*

① *When the sweep-line is located at instant $\delta_t$ we can start task $t$ at $min_t$ without exceeding $limit$, i.e.*

$$\forall t' \in \mathcal{T}\backslash\{t\}, \forall i \in [min_t, \delta_t) : \; h_t + \sum_{\substack{t' \in \mathcal{T}\backslash\{t\},\\ i \in [\overline{s_{t'}}, \underline{e_{t'}})}} h_{t'} \leq limit$$

*The adjustment of the earliest start of task $t$ to $min_t$ implies that task $t$ is not in conflict on the interval $[min_t, \delta_t)$ wrt. the CPP. Condition* $\mathrm{get\_top\_key}(h\_check) > gap$ *(Algorithm 2, line 4) ensures that the adjustment in line 7 does not induce a resource overflow on $[min_t, \delta_t)$, otherwise $t$ should have been added into $h\_conflict$. Condition* $\mathrm{get\_top\_key}(h\_conflict) \leq gap$ *(Algorithm 2, line 12) implies that task $t$ is in conflict until the current sweep-line position $\delta$. If $\delta \geq \overline{s_t}$ (line 14) the conflict on $[\overline{s_t}, \delta_t)$ is not "real" since the compulsory part of $t$ is already taken into account in the CPP. In line 19 of Algorithm 2, the earliest start of task $t$ is adjusted to the current sweep-line position, consequently the interval $[min_t, \delta_t)$ is empty.*

② *For each value of $\delta$ greater than $\delta_t$,* sweep_min *cannot create a compulsory part before instant $\delta_t$. This is implied by Lemma 1, which ensures that* sweep_min *cannot generate any compulsory part before $\delta$.*

*Consequently once* sweep_min *is completed, any task $t$ can be fixed to its earliest start without creating a CPP exceeding the resource limit $limit$.* □

## 3.5 Complexity

Given a *cumulative* constraint involving $n$ tasks, the worst-case time complexity of the dynamic sweep algorithm is $O(n^2 \log n)$. First note that the overall worst-case complexity of synchronize over a full sweep is $O(n)$ since conditional and dynamic events are updated at most once. The worst-case $O(n^2 \log n)$ can be reached in the special case when the CPP consists of a succession of high peaks and deep, narrow valleys.

Assume that one has $O(n)$ peaks, $O(n)$ valleys, and $O(n)$ tasks to switch between $h\_check$ and $h\_conflict$ each time. A heap operation costs $O(\log n)$. The resulting worst-case time complexity is $O(n^2 \log n)$. For bin-packing, the two heaps $h\_conflict$ and $h\_check$ permit to reduce the worst-case time complexity down to $O(n \log n)$. Indeed, the earliest start of the tasks of duration one that exit $h\_conflict$ can directly be adjusted (i.e. $h\_check$ is unused).

# 4 A Synchronized Sweep Algorithm for the *k-dimensional cumulative* Constraint

This section presents a new synchronized sweep algorithm that handles several cumulative resources in one single sweep. In this new setting, each task uses several cumulative resources and the challenge is to come with an approach that scales well. We should quote that the number of resources may be significant in many situations:

- For instance, in the 2012 Roadef Challenge we have up to 12 distinct resources per item to pack.
- A new resource $r'$ can also be introduced for modeling the fact that a given subset of tasks is subject to a *cumulative* or *disjunctive* constraint. The tasks that do not belong to the subset have their consumption of resource $r'$ set to 0. This is indeed the case for the industrial application presented in the evaluation section. Since we potentially can have a lot of such constraints on different subsets of tasks, this can lead to a large number of resources.

This new synchronized sweep algorithm is an efficient scalable $k$-dimensional version of the *timetable* method which achieves exactly the same pruning as $k$ instances of the 1-dimensional version reported in Section 3. Note that this version differs from the one introduced in [18], and despite the fact that it scales a little worse when considering the number of tasks, it scales a lot better when considering the number of resources.

Given $k$ resources and $n$ tasks, where each resource $r$ ($0 \leq r < k$) is described by its maximum capacity $limit_r$, and each task $t$ ($0 \leq t < n$) is described by its start $s_t$, fixed duration $d_t$ ($d_t \geq 0$), end $e_t$ and fixed resource consumptions $h_{t,0}, \ldots, h_{t,k-1}$ ($h_{t,i} \geq 0$, $i \in [0, k-1]$) on the $k$ resources, the *k-dimensional cumulative* constraint with the two arguments

- $\langle\langle s_0, d_0, e_0, \langle h_{0,0}, \ldots, h_{0,k-1}\rangle\rangle, \ldots, \langle s_{n-1}, d_{n-1}, e_{n-1}, \langle h_{n-1,0}, \ldots, h_{n-1,k-1}\rangle\rangle\rangle$,
- $\langle limit_0, \ldots, limit_{k-1}\rangle$

holds if and only if conditions (4) and (5) are both true:

$$\forall t \in [0, n-1] \ : s_t + d_t = e_t \tag{4}$$

$$\forall r \in [0, k-1],\ \forall i \in \mathbb{Z} \ : \sum_{\substack{t \in [0,n-1],\\ i \in [s_t, e_t)}} h_{t,r} \leq limit_r \tag{5}$$

**Example 2** *(Example 1 extended with an extra resource $r_1$) Consider two resources $r_0$, $r_1$ ($k = 2$) with $limit_0 = 3$ and $limit_1 = 2$ and five tasks $t_0$,$t_1$,...,$t_4$ which have the following restrictions on their* start, duration, end *and* heights*:*

- $t_0$: $s_0 \in [1,1]$, $d_0 = 1$, $e_0 \in [2,2]$, $h_{0,0} = 2$, $h_{0,1} = 1$
- $t_1$: $s_1 \in [0,3]$, $d_1 = 2$, $e_1 \in [2,5]$, $h_{1,0} = 2$, $h_{1,1} = 1$
- $t_2$: $s_2 \in [0,5]$, $d_2 = 2$, $e_2 \in [2,7]$, $h_{2,0} = 1$, $h_{2,1} = 2$
- $t_3$: $s_3 \in [0,9]$, $d_3 = 1$, $e_3 \in [1,10]$, $h_{3,0} = 1$, $h_{3,1} = 1$
- $t_4$: $s_4 \in [0,7]$, $d_4 = 3$, $e_4 \in [3,10]$, $h_{4,0} = 2$, $h_{4,1} = 0$

*Since task $t_1$ cannot overlap $t_0$ without exceeding the resource limit on resource $r_0$, the earliest start of $t_1$ is adjusted to 2. Since $t_1$ occupies the interval $[3, 4)$ and since, on resource $r_0$, $t_4$ cannot overlap $t_1$, its earliest start is adjusted to 4. On resource $r_1$, since $t_2$ cannot overlap task $t_1$, its earliest start is adjusted to 4. The purpose of the synchronized sweep algorithm is to perform such filtering in an efficient way, i.e. in one single sweep.* □

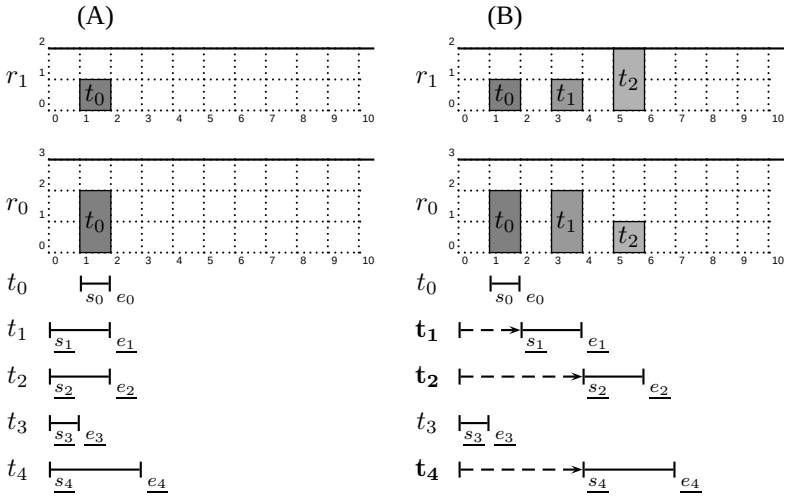


Figure 2: Parts (A) and (B) respectively represent the earliest positions of the tasks and the CPP on resource $r_0$ and $r_1$, (A) of the initial problem described in Example 2, (B) once the fixpoint is reached.

We now show how decomposing the *2-dimensional cumulative* constraint into two *cumulative* constraints on resource $r_0$ and $r_1$ leads to a ping-pong between the two constraints to reach the fixpoint.

*Continuation of Example 2 (Illustrating the ping-pong induced by the decomposition).* The instance given in Example 2 can naturally be decomposed into two *cumulative* constraints:

- $c_0 : cumulative(\langle \quad \langle s_0, d_0, e_0, h_{0,0}\rangle, \langle s_1, d_1, e_1, h_{1,0}\rangle, \langle s_2, d_2, e_2, h_{2,0}\rangle, \langle s_3, d_3, e_3, h_{3,0}\rangle, \langle s_4, d_4, e_4, h_{4,0}\rangle\rangle, limit_0),$
- $c_1 : cumulative(\langle \quad \langle s_0, d_0, e_0, h_{0,1}\rangle, \langle s_1, d_1, e_1, h_{1,1}\rangle, \langle s_2, d_2, e_2, h_{2,1}\rangle, \langle s_3, d_3, e_3, h_{3,1}\rangle\rangle, limit_1).$ .

- During a first sweep wrt. constraint $c_0$ (see Part (A) of Figure 3), the compulsory part of the task $t_0$ on resource $r_0$ and on interval $[1, 2)$ permits to adjust the earliest start of task $t_1$ to 2 since the gap on top of this interval is strictly less than the resource consumption of $t_1$ on $r_0$. Task $t_1$ now has a compulsory part on the interval $[3, 4)$. This new compulsory part permits to adjust the earliest start of the task $t_4$ to 4.
- A second sweep wrt. constraint $c_1$ (see Part (B) of Figure 3), adjusts the earliest start of task $t_2$ since it cannot overlap neither the compulsory part of task $t_0$ nor the compulsory part of task $t_1$. So task $t_2$ now has a compulsory part on the interval $[5, 6)$.
- Finally a third sweep wrt. constraint $c_0$ is performed to find out that nothing more can be deduced and that the fixpoint is reached. □

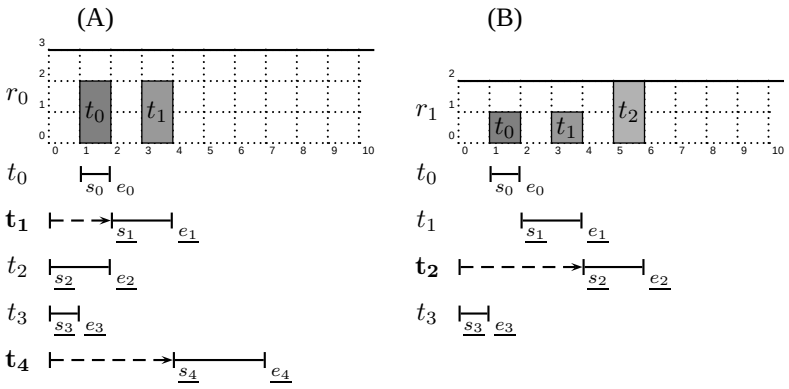


Figure 3: Parts (A) and (B) respectively represent the earliest positions of the tasks and the CPP, after a first sweep on the resource $r_0$, and after a second sweep on $r_1$.

Our new *sweep_min* filtering algorithm will perform such deductions in one single step.

We now give the fixpoint property achieved by our new *sweep_min* algorithm that handles the *k-dimensional cumulative* constraint.

**Property 2** *Given a k-*dimensional cumulative *constraint with $n$ tasks and $k$ resources, the corresponding* sweep_min *algorithm ensures that:*

$$\forall r \in [0, k-1], \forall t \in [0, n-1], \forall i \in [\underline{s_t}, \underline{e_t}) \ : \ h_{t,r} + \sum_{\substack{t' \neq t, \\ i \in [\overline{s_{t'}}, \underline{e_{t'}})}} h_{t',r} \leq limit_r \tag{6}$$

Property 2 ensures that, for any task $t$ of the *k-dimensional cumulative* constraint, one can schedule $t$ at its earliest start without exceeding for any resource $r$ ($0 \leq r < k$) its resource limit wrt. the CPP on resource $r$ of the tasks of $\mathcal{T} \backslash \{t\}$.

## 4.1 Event Point Series

Since events are only related to the temporal aspect, they do not depend on how many resources we have, and can therefore be factored out. The only difference with the event point series of [18] is that the $CCP$ event type has been merged with the $SCP$ event type. This is possible since they are related to the same time point, i.e. the latest start of a task. In order to build the CPP on each resource and to filter the earliest start of each task, the algorithm considers the following types of events.

- The event type $\langle SCP, t, \overline{s_t} \rangle$ for the *Start of Compulsory Part* of task $t$ (i.e. the latest start of task $t$). This event is generated for all the tasks. If the task has no compulsory part when the event is read, it will simply be ignored.
- The event type $\langle ECPD, t, \underline{e_t} \rangle$ where the date of such event corresponds to the *End of the Compulsory Part* of task $t$ (i.e. the earliest end of task $t$) and may increase due to the adjustment of the earliest start of $t$. This event is generated if and only if task $t$ has a compulsory part, i.e. if and only if $\overline{s_t} < \underline{e_t}$.
- The event type $\langle PR, t, \underline{s_t} \rangle$ where $PR$ stands for *Pruning Event*, corresponds to the earliest start of task $t$. This event is generated if and only if task $t$ is not yet scheduled, i.e. if and only if $\underline{s_t} \neq \overline{s_t}$.

As in the single resource case, events are recorded in the heap $h\_events$ where the top event is the event with the smallest date.

*Continuation of Example 2 (Generated Events).* The following events are generated and sorted according to their date: $\langle PR, 1, 0\rangle$, $\langle PR, 2, 0\rangle$, $\langle PR, 3, 0\rangle$, $\langle PR, 4, 0\rangle$, $\langle SCP, 0, 1\rangle$, $\langle ECPD, 0, 2\rangle$, $\langle SCP, 1, 3\rangle$, $\langle SCP, 2, 5\rangle$, $\langle SCP, 3, 9\rangle$, $\langle SCP, 4, 7\rangle$. □

## 4.2 Sweep-Line Status

In order to build the CPP and to filter the earliest start of the tasks, the sweep-line jumps from event to event, maintaining the following information:

- The current sweep-line position $\delta$, initially set to the date of the first event.
- For each resource $r \in [0, k-1]$, the amount of available resource at instant $\delta$ denoted by $gap_r$ (i.e. the difference between the resource limit $limit_r$ and the height of the CPP on resource $r$ at instant $\delta$) and its previous value denoted by $gap'_r$.
- For each task $t \in [0, n-1]$, $ring_t$ stores its status, and is equal to:
  - $none$ if and only if the sweep-line has not yet read the $PR$ event related to task $t$,
  - $ready$ if and only if the earliest start of task $t$ was adjusted to its final value (i.e. the fixpoint was reached for the earliest start of task $t$),
  - $check$ if and only if $\delta \in [\underline{s_t}, \overline{s_t})$ and $\forall r \in [0, k-1] : h_{t,r} \leq gap_r$, i.e. for all resources, the resource consumption of task $t$ does not exceed the available gap on top of the corresponding CPP,
  - $conflict_r$ if and only if $\delta \in [\underline{s_t}, \overline{s_t})$ and $\exists r \in [0, k-1) : h_{t,r} > gap_r$, i.e. there is at least one resource $r$ where task $t$ is in conflict. Note that we only record the first resource where there is a conflict.

  All tasks $t$ for which $ring_t = check$ or $ring_t = conflict_r$ are called *active tasks* in the following. From an implementation point of view, the status of the active tasks are stored in rings, i.e. circular double linked lists, which permits us to quickly iterate over all tasks in $check$ or $conflict$ status, as well as to move in constant time a task from $check$ to $conflict$ status or vice versa. In the following, $conflict_\star$ is used to indicate that task $t$ is in conflict on a resource whose identifier we don't need to know.

Our synchronized sweep algorithm first creates and sorts the events wrt. their date. Then, the sweep-line moves from one event to the next event, updating the amount of available space on each resource (i.e. $gap_r,\ 0 \leq r < k$), and the status of the tasks accordingly. Once all events at instant $\delta$ have been handled, the algorithm tries to filter the earliest start of the active tasks wrt. $gap_r$ ($0 \leq r < k$) and to the sweep interval $[\delta, \delta_{next})$, where $\delta_{next}$ is the next sweep-line position. In order to update the status of the tasks, for each resource $r$, if $gap_r$ has decreased compared to the gap at the previous sweep-line position, we scan all the tasks $t$ that potentially can switch their status to $conflict$ or $ready$ (i.e. all tasks $t$ for which $ring_t = check$). Symmetrically, for each resource $r$, if $gap_r$ has increased, we scan all the tasks that are potentially no longer in $conflict_r$.

## 4.3 Algorithm

The *sweep_min* part of the synchronized sweep algorithm consists of a main loop (Algorithm 4), a processing events part (Algorithm 5) and a filtering part (Algorithm 6).

#### 4.3.1 Main Loop

The main loop (Algorithm 4) consists of:

- `[CREATING EVENTS]` (line 2). The events are generated wrt. the start and end variables of each task and inserted into the heap $h\_events$, which records the events sorted by increasing date.
- `[INITIALIZATION]` (lines 4 to 7). The available space $gap_r$ and the previous available space $gap'_r$ of each resource $r$ is set the corresponding resource limit $capa_r$. For each task $t$ its status is set to $none$ if $t$ is not fixed, $ready$ otherwise.
- `[MAIN LOOP]` (lines 9 to 12). For each sweep-line position the main loop processes all the corresponding events and updates the sweep-line status. In this last part, Algorithm 4 returns **false** if a resource overflow occurs.

**ALGORITHM** sweep_min() : boolean
1: `[CREATING EVENTS]`
2: $h\_events \leftarrow$ generation of events wrt. $n,\ \underline{s_t},\ \overline{s_t},\ d_t,\ \underline{e_t}$.
3: `[INITIALIZATION]`
4: **for** $r = 0$ **to** $k - 1$ **do**
5: $\quad gap_r, gap'_r \leftarrow capa_r$
6: **for** $t = 0$ **to** $n - 1$ **do**
7: $\quad$ **if** $\underline{s_t} = \overline{s_t}$ **then** $ring_t \leftarrow ready$ **else** $ring_t \leftarrow none$
8: `[MAIN LOOP]`
9: **while** $\neg$empty$(h_events)$ **do**
10: $\quad \langle \delta, \delta_{next} \rangle \leftarrow$ process_events()
11: $\quad$ **if** $\neg$filter_min$(\delta, \delta_{next})$ **then**
12: $\quad\quad$ **return false**
13: **return true**

**Algorithm 4:** Main sweep algorithm. Returns **false** if a resource overflow occurs, **true** otherwise. If **true**, ensures that the earliest start of each task is pruned so that Property 2 holds.

#### 4.3.2 The Event Processing Part

In order to update the sweep-line status, Algorithm 5 reads and processes all the events related to the current sweep-line position $\delta$ and determines the sweep interval $[\delta, \delta_{next})$. Algorithm 5 consists in the following parts:

- `[PROCESSING START COMPULSORY PART (SCP) EVENTS]` (lines 3 to 14). When the sweep-line reaches the latest start of a task $t$, we have to determine whether or not the earliest start of task $t$ can still be updated. This requires the following steps to be considered:
  - If task $t$ is in conflict (i.e. $ring_t = conflict_\star$, line 5), then $t$ cannot be scheduled before its latest position.
  - If the status of task $t$ is $check$ (line 8), meaning that there is no conflict on the interval $[\underline{s_t}, \overline{s_t})$, then the earliest start of $t$ cannot be updated. To ensure Property 2, the consumption of task $t$ on the interval $[\overline{s_t}, \underline{e_t})$, which is empty if task $t$ has no compulsory part, is taken into account in the CPP.

  Once the earliest start and end of the task are up to date, we need to know whether a compulsory part was created for task $t$ (i.e., whether $\delta = \overline{s_t}$ is strictly less than $\underline{e_t}$, line 10). If a compulsory part has appeared, the gaps are decreased accordingly and an $ECPD$ event is added into $h\_events$.

- `[PROCESSING DYNAMIC (ECPD) EVENTS]` (lines 16 to 21). When the sweep-line reaches the $ECPD$ event of a task $t$ we first have to check that the date of this event is well placed wrt. the sweep-line. If not ($\underline{e_t} > \delta$, line 17), the $ECPD$ event is pushed back into the heap $h\_event$ to its correct date (line 18). If the event is well placed, the available spaces are updated (lines 20 to 21).
- `[DETERMINE THE NEXT EVENT DATE]` (line 23). In order to process the pruning ($PR$) events, we first need to know the next position $\delta_{next}$ of the sweep-line.
- `[PROCESSING EARLIEST START (PR) EVENTS]` (lines 25 to 31). If a conflict is detected (i.e. $\exists r \mid h_{t,r} > gap_r$, line 26) the status of the task $t$ is set to $conflict$. Else if the sweep interval is too small wrt. the duration of task $t$ (i.e. $\underline{e_t} > \delta_{next}$), the status of $t$ is set to $check$. Else we know that the earliest start of task $t$ cannot be further adjusted wrt. Property 2

**ALGORITHM** process_events() : $\langle \text{integer}, \text{integer} \rangle$

1: $\langle \delta, \mathcal{E} \rangle \leftarrow$ extract and record in $\mathcal{E}$ all the events in $h\_events$ related to the minimal date $\delta$
2: `[PROCESSING START COMPULSORY PART (SCP) EVENTS]`
3: **for all** events of type $\langle SCP, t, \overline{s_t} \rangle$ in $\mathcal{E}$ **do**
4: $ecp' \leftarrow \underline{e_t}$
5: **if** $ring_t = \text{conflict}_\star$ **then**
6: adjust_min_var$(s_t, \overline{s_t})$; adjust_min_var$(e_t, \overline{e_t})$;
7: $ring_t \leftarrow \text{ready}$
8: **else if** $ring_t = \text{check}$ **then**
9: $ring_t \leftarrow \text{ready}$
10: **if** $\delta < \underline{e_t}$ **then**
11: **for** $r = 0$ **to** $k - 1$ **do**
12: $gap_r \leftarrow gap_r - h_{t,r}$
13: **if** $ecp' \leq \delta$ **then** // introduce $ECPD$ event if new CP
14: add $\langle ECPD, t, \underline{e_t} \rangle$ to $h\_events$
15: `[PROCESSING DYNAMIC (ECPD) EVENTS]`
16: **for all** events of type $\langle ECPD, t, \underline{e_t} \rangle$ in $\mathcal{E}$ **do**
17: **if** $\underline{e_t} > \delta$ **then** // reintroduce $ECP$ event if $\underline{e_t}$ has moved
18: add $\langle ECPD, t, \underline{e_t} \rangle$ to $h\_events$
19: **else**
20: **for** $r = 0$ **to** $k - 1$ **do**
21: $gap_r \leftarrow gap_r + h_{t,r}$
22: `[DETERMINE THE NEXT EVENT DATE]`
23: $\delta_{next} \leftarrow$ get_top_key$(h_events)$ // $+\infty$ if empty
24: `[PROCESSING EARLIEST START (PR) EVENTS]`
25: **for all** events of type $\langle PR, t, \underline{s_t} \rangle$ in $\mathcal{E}$ **do** // $PR$ must be handled last
26: **if** $\exists r \mid h_{t,r} > gap_r$ **then** // is task $t$ in conflict?
27: $ring_t \leftarrow \text{conflict}_r$
28: **else if** $\underline{e_t} > \delta_{next}$ **then** // might task $t$ be in conflict next time?
29: $ring_t \leftarrow \text{check}$
30: **else**
31: $ring_t \leftarrow \text{ready}$
32: **return** $\langle \delta, \delta_{next} \rangle$

**Algorithm 5:** Called every time the sweep-line moves. Extracts and processes all events at given time point $\delta$. Returns both the current $\delta$ and the next time point $\delta_{next}$.

#### 4.3.3 The Filtering Part

Algorithm 6 takes into account the variation of the gaps on top of the CPP between the previous and the current position of the sweep-line in order to process tasks that are in $conflict$ or in $check$ status and

possibly to adjust their earliest start. The main parts of the algorithm are:

- `[CHECK RESOURCE OVERFLOW]` (lines 2 to 3). If the available resource is negative on at least one resource on the sweep interval $[\delta, \delta_{next})$, Algorithm 6 returns **false** for failure.
- `[TASKS NO LONGER IN CHECK]` (lines 5 to 9). Scans each resource $r$ where the current available resource is less than the previous available space (i.e. $gap'_r > gap_r$, line 6). It has to consider each task $t$ which is in $check$ such that the height of task $t$ is greater than the current available space (line 7), i.e. tasks which are no longer in $check$. If the sweep-line has passed the earliest end of task $t$, meaning that there is no conflict on the interval $[\underline{s_t}, \underline{e_t})$ its status is updated to $ready$. Otherwise, the status of task $t$ is set to $check$.
- `[TASKS NO LONGER IN CONFLICT ON RESOURCE` $r$`]` (lines 11 to 22). Scans each resource $r$ where the current available resource is strictly greater than the previous available resource. It has to consider each task $t$ which is in conflict such that the height of $t$ is less than or equal to the current available space (line 13), i.e. tasks which are no longer in conflict on resource $r$. We consider the two following cases:
  - If the task $t$ is in conflict on another resource $r'$ ($\exists r'.h_{t,r'} > gap_{r'}$, line 14) its status is set to $conflict$.
  - Otherwise, the earliest start of task $t$ is updated (line 18) to the current sweep-line position and an $ECPD$ event is added if a new compulsory part occurs (lines 20 to 21).

**ALGORITHM** $\text{filter\_min}(\delta, \delta_{next}) : \text{boolean}$

```
1:  [CHECK RESOURCE OVERFLOW]
2:  for r = 0 to k − 1 do                                   // fail if capacity exceeded
3:    if gap_r < 0 then return false
4:  [TASKS NO LONGER IN CHECK]
5:  for r = 0 to k − 1 do
6:    if gap'_r > gap_r then
7:      for all t | ring_t = check ∧ h_{t,r} > gap_r do
8:        ring_t ← if e_t > δ then conflict_r else ready
9:      gap'_r ← gap_r
10: [TASKS NO LONGER IN CONFLICT ON RESOURCE r]
11: for r = 0 to k − 1 do
12:   if gap'_r < gap_r then
13:     for all t | ring_t = conflict_r ∧ h_{t,r} ≤ gap_r do
14:       if ∃r'.h_{t,r'} > gap_{r'} then
15:         ring_t ← conflict_{r'}
16:       else
17:         ecp' ← e_t
18:         adjust_min_var(s_t, δ); adjust_min_var(e_t, δ + d_t);
19:         ring_t ← if e_t > δ_next then check else ready
20:         if s_t ≥ ecp' ∧ s_t < e_t then                  // introduce ECPD event if new compulsory part
21:           add ⟨ECPD, t, e_t⟩ to h_events
22:     gap'_r ← gap_r
23: return true
```

**Algorithm 6:** Called every time the sweep-line moves from $\delta$ to $\delta_{next}$ in order to try to filter the earliest start of the tasks wrt. the available space on each resource.

*Continuation of Example 2 (Illustrating the Synchronized Sweep Algorithm).* The synchronized sweep algorithm first initializes the current sweep-line position to 0, $gap_0$ to 3, and $gap_1$ to 2. Since task $t_0$ is fixed, its status is set to $ready$, and to $none$ for all the other tasks (Algorithm 4, line 7). The sweep-line reads the four $PR$ events related to the tasks $t_1$, $t_2$, $t_3$ and $t_4$. The next event date permits to set $\delta_{next}$ to

1. On the one hand the status of the tasks $t_1$, $t_2$ and $t_4$ is set to $check$ since their heights on both resources are less than or equal to the corresponding available spaces and since their duration is strictly greater than the size of the sweep interval $[0, 1)$ (Algorithm 5, line 29). On the other hand the status of task $t_3$ is set to $ready$ since its duration is less than or equal to the size of the sweep interval. The first call of $\mathrm{filter\_min}$ does not deduce anything since $gap_0$ and $gap_1$ are respectively equal to ${gap'}_0$ and ${gap'}_1$. Then it moves the sweep-line to position 1, reads the $SCP$ event related to task $t_0$ and reads the next event date 2. During $\mathrm{filter\_min}$, status of tasks $t_1$ and $t_4$ are set to $conflict$ because of their too big height on resource $r_0$. The status of task $t_2$ is also set to $conflict$, because of its too big height on resource $r_1$ (Algorithm 6, line 8). Then it moves the sweep-line to position 2, reads the $ECPD$ event related to $t_0$ and reads the next event date 3. During $\mathrm{filter\_min}$, the earliest start of tasks $t_4$, $t_1$ and $t_2$ is set to 2 and their status is set to $check$. Moreover the following $ECPD$ event is generated for $t_1$, $\langle ECPD, 1, 4\rangle$ (Algorithm 6, lines 18 to 21). Then it moves the sweep-line to position 3 and reads the $SCP$ event of task $t_1$. Since the current status of $t_1$ is $check$, we know that the earliest start of task $t_1$ cannot be adjusted anymore and consequently the status of task $t_1$ is set to $ready$ (Algorithm 5, line 9). The next event date is 4. The call to $\mathrm{filter\_min}$ on the sweep interval $[3, 4)$ with $gap_0 = 1$ and $gap_1 = 1$ changes the status of tasks $t_2$ and $t_4$ to $conflict$ because $h_{2,1} > gap_1$ and $h_{4,0} > gap_0$ (Algorithm 6, line 8). Then it moves the sweep-line to position 4, reads the $ECPD$ event of task $t_1$ and set $gap_0$ to 3 and $gap_1$ to 2. During $\mathrm{filter\_min}$, since the available spaces increase, the status of tasks $t_2$ and $t_4$ change from $conflict$ to $check$, their earliest start is adjusted to 4, and the following $ECPD$ event is generated for task $t_2$, $\langle ECPD, 2, 6\rangle$ (Algorithm 6, lines 18 to 21). No $ECPD$ event is generated for task $t_4$ since its earliest end is always less than or equal to its latest start (Algorithm 6, lines 18). Then the sweep-line reads the $SCP$ and $ECPD$ events related to task $t_2$, nothing more can be deduced, and Property 2 is holds. □

### 4.4 Complexity

Given a *k-dimensional cumulative* involving $n$ tasks, the worst-case time complexity of the synchronized sweep algorithm is $O(kn^2)$. Initially, at most three events are generated per task. In addition, at most one dynamic $ECPD$ event can be generated per task. Since one event is handled in $O(k + \log n)$, the overall worst-case time complexity of Algorithm 5 over a full sweep is $O(kn+n \log n)$. Like for the 1-dimensional dynamic sweep, the worst-case time complexity is reached when the CPP consists of a succession of high peaks and deep, narrow valleys. In this worst-case, Algorithm 6 has to change the status of the $n$ tasks, which is done in $O(kn)$ since lines 14 to 21 are executed at most once per task and since line 14 costs $O(k)$. Algorithm 6 is called at each step of the sweep, which result in a complexity of $O(kn^2)$.

## 5 A Synchronized Sweep Algorithm for the *k-dimensional cumulative with precedences* Constraint

This section presents an extension of the synchronized sweep algorithm introduced in Section 4 that also handles a set of precedence constraints among the tasks. In this context, a precedence between a task $t$ and a task $t'$ means that task $t$ must be completed before task $t'$ starts, i.e. $s_t + d_t \leq e_{t'}$. Our goal is to provide an algorithm that scales well, even with a high number of precedence constraints, which is usually a source of inefficiency in CP solvers (see Point ⑤ `[Too Local]` of Section 2).

Given $k$ resources and $n$ tasks, where each resource $r$ ($0 \leq r < k$) is described by its maximum capacity $limit_r$, where each task $t$ ($0 \leq t < n$) has a list of successors $P_t$ and is described by its start $s_t$, fixed duration $d_t$ ($d_t > 0$), end $e_t$, fixed resource consumptions $h_{t,0}, \ldots, h_{t,k-1}$ ($h_{t,i} \geq 0$, $i \in [0, k-1]$) on the $k$ resources, the *k-dimensional cumulative with precedences* constraint with the three arguments

- $\langle\langle s_0, d_0, e_0, \langle h_{0,0}, \ldots, h_{0,k-1}\rangle\rangle, \ldots, \langle s_{n-1}, d_{n-1}, e_{n-1}, \langle h_{n-1,0}, \ldots, h_{n-1,k-1}\rangle\rangle\rangle$,
- $\langle limit_0, \ldots, limit_{k-1}\rangle$
- $\langle P_0, \ldots, P_{n-1}\rangle$

holds if and only if conditions (7), (8) and (9) are true:

$$\forall t \in [0, n-1] \ : s_t + d_t = e_t \tag{7}$$

$$\forall r \in [0, k-1], \ \forall i \in \mathbb{Z} \ : \sum_{\substack{t \in [0,n-1], \\ i \in [s_t, e_t)}} h_{t,r} \leq limit_r \tag{8}$$

$$\forall t \in [0, n-1], \ \forall t' \in P_t \ : e_t \leq s_{t'} \tag{9}$$

Note that the graph of precedences is supposed to be acyclic.

**Example 3** *(Example 2 extended with precedence constraints) Consider two resources $r_0$, $r_1$ ($k = 2$) with $limit_0 = 3$ and $limit_1 = 2$ and five tasks $t_0$,$t_1$,...,$t_4$ which have the following restrictions on their* start, duration, end *and* heights*:*

- $t_0$: $s_0 \in [1,1]$, $d_0 = 1$, $e_0 \in [2,2]$, $h_{0,0} = 2$, $h_{0,1} = 1$
- $t_1$: $s_1 \in [0,3]$, $d_1 = 2$, $e_1 \in [2,5]$, $h_{1,0} = 2$, $h_{1,1} = 1$
- $t_2$: $s_2 \in [0,5]$, $d_2 = 2$, $e_2 \in [2,7]$, $h_{2,0} = 1$, $h_{2,1} = 2$
- $t_3$: $s_3 \in [0,9]$, $d_3 = 1$, $e_3 \in [1,10]$, $h_{3,0} = 1$, $h_{3,1} = 1$
- $t_4$: $s_4 \in [0,7]$, $d_4 = 3$, $e_4 \in [3,10]$, $h_{4,0} = 2$, $h_{4,1} = 0$

*We also consider the following three precedence constraints among the tasks:*

- $e_0 \leq s_3$*, meaning that task $t_0$ has to end before task $t_3$ starts,*
- $e_1 \leq s_3$*, meaning that task $t_1$ has to end before task $t_3$ starts,*
- $e_2 \leq s_4$*, meaning that task $t_2$ has to end before task $t_4$ starts.*

*On the one hand, if we ignore the precedence constraints we have the same instance than Example 2, consequently we have the same pruning, i.e. because of the resource constraint on $r_0$ and $r_1$, the earliest start of task $t_1$ is adjusted to 2, the earliest start of tasks $t_2$ and $t_4$ is adjusted to 4 (see Part (B), Figure 2). On the other hand, considering the precedence constraints leads to the following additional adjustments: the earliest start of task $t_3$ is adjusted to 4 since the earliest end of task $t_1$ is 4 and the earliest start of task $t_4$ is adjusted to 6 since the earliest end of task $t_2$ is 6 (see Part (B), Figure 4). The purpose of the synchronized sweep algorithm, extended to handle a set of precedence constraints, is to perform such filtering wrt. all resource and precedence constraints in one single sweep.* □

We now show how to achieve such filtering by decomposing the *2-dimensional cumulative with precedences* constraint into two *cumulative* constraint on resource $r_0$ and $r_1$ and three precedence constraints.

*Continuation of Example 2 (Illustrating the decomposition).* The instance given in Example 3 can naturally be decomposed into two *cumulative* constraints and three inequality constraints:

- $c_0 : cumulative(\langle \ \langle s_0, d_0, e_0, h_{0,0} \rangle, \langle s_1, d_1, e_1, h_{1,0} \rangle, \langle s_2, d_2, e_2, h_{2,0} \rangle, \langle s_3, d_3, e_3, h_{3,0} \rangle, \langle s_4, d_4, e_4, h_{4,0} \rangle \rangle, limit_0)$ .
- $c_1 : cumulative(\langle \ \langle s_0, d_0, e_0, h_{0,1} \rangle, \langle s_1, d_1, e_1, h_{1,1} \rangle, \langle s_2, d_2, e_2, h_{2,1} \rangle, \langle s_3, d_3, e_3, h_{3,1} \rangle \rangle, limit_1)$ .
- $c_3 : e_0 \leq s_3$.

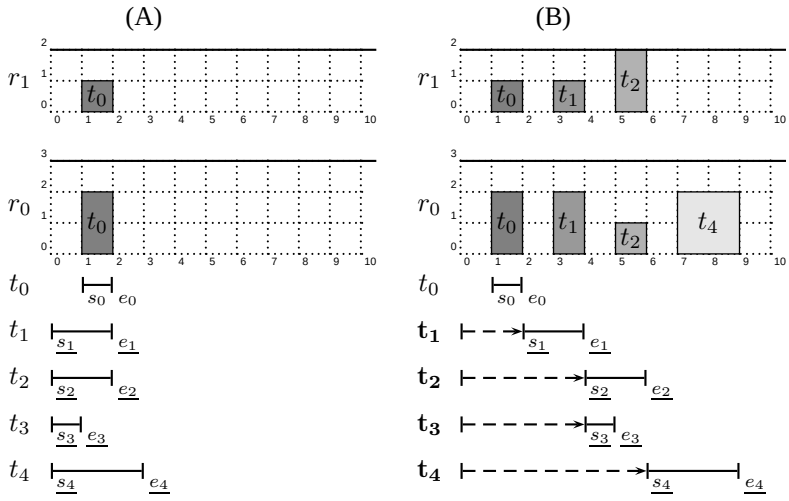


Figure 4: Parts (A) and (B) respectively represent the earliest positions of the tasks and the CPP on resource $r_0$ and $r_1$, (A) of the initial problem described in Example 3, (B) once the fixpoint is reached.

- $c_4 : e_1 \leq s_3$.
- $c_5 : e_2 \leq s_4$.

Traditionally, a CP solver will first process the lightest constraints, i.e. $c_3$, $c_4$ and $c_5$ and reach a fixpoint over this subset of constraints. Then, it will process one *cumulative* constraint. These two steps are repeated until the fixpoint over the five constraints is reached. The source of inefficiency comes from the fact that when a precedence constraint prunes one variable, we need to rerun from scratch all the cumulative constraints involving the corresponding variable. □

The property ensures by the *sweep_min* algorithm is an extension of Property 2 that also considers the precedence constraints.

**Property 3** *Given a k-*dimensional cumulative with precedences *constraint with $n$ tasks and $k$ resources,* sweep_min *ensures that:*

$$\forall r \in [0, k-1], \forall t \in [0, n-1], \forall i \in [\underline{s_t}, \underline{e_t}) \ : \ h_{t,r} + \sum_{\substack{t' \neq t, \\ i \in [\underline{s_{t'}}, \underline{e_{t'}})}} h_{t',r} \leq limit_r \tag{10}$$

$$\forall t \in [0, n-1], \forall t' \in P_t \ : \ \underline{e_t} \leq \underline{s_{t'}} \tag{11}$$

Property 3 ensures that, for any task $t$ of the *k-dimensional cumulative with precedences* constraint, one can schedule $t$ at its earliest start without exceeding for any resource $r$ ($0 \leq r < k$) its resource limit wrt. the CPP on resource $r$ of the tasks of $\mathcal{T} \backslash \{t\}$, and all its immediate successors cannot start before its earliest end.

## 5.1 Event Point Series

In order to build the CPP of the resources, we need all the event types required by the synchronized sweep algorithm for the *k-dimensional cumulative* constraint (see Section 4.1). To ensure Relation 11 of Property 3, all the events related to tasks that have at least one predecessor are not initially added into the heap of events. A task will only be added when the earliest starts of all its immediate predecessors are adjusted to their final position wrt Property 3. More precisely, to know the moment when these events must be added, we introduce the following new event type:

- The event type $\langle RS, t, \underline{e_t} \rangle$ for *Release Successors* of task $t$ (i.e. the earliest end of task $t$) is generated for all the tasks $t$ that have at least one successor. This is required to prevent the earliest starts of the successors of task $t$ from being adjusted before the final earliest start of task $t$ has been determined.

*Continuation of Example 3 (Generated Events).* In the initialization part, the following events are generated and sorted according to their date: $\langle PR, 1, 0 \rangle$, $\langle PR, 2, 0 \rangle$, $\langle SCP, 0, 1 \rangle$, $\langle ECPD, 0, 2 \rangle$, $\langle \mathbf{RS}, \mathbf{0}, \mathbf{2} \rangle$, $\langle \mathbf{RS}, \mathbf{1}, \mathbf{2} \rangle$, $\langle \mathbf{RS}, \mathbf{2}, \mathbf{2} \rangle$, $\langle SCP, 1, 3 \rangle$, $\langle SCP, 2, 5 \rangle$. On the one hand, since tasks $t_0$, $t_1$ and $t_2$ all have at least one successor we generate one $RS$ event, in bold, for each of them. On the other hand since tasks $t_3$ and $t_4$ do not have any successor, we do not generate any $RS$ events for them. □

## 5.2 Sweep-Line Status

All the elements of the sweep-line status of the synchronized sweep are needed to build the CPP over the resources. In addition, we introduce the following information to handle the precedence constraints among the tasks:

- For each task $t \in [0, n-1]$, $nbpred_t$ records the number of predecessors of task $t$ for which the final value of the earliest start wrt. Property 3 was not yet found at the current sweep-line position $\delta$.

The synchronized sweep algorithm with precedences first creates and sorts the events wrt. their date for the tasks that have no predecessors. Then the sweep-line moves from one event to the next event, updating the amount of available space on each resource and the status of the tasks. Only once the earliest starts of all the predecessors of a given task $t$ have been found, i.e. $nbpred_t = 0$, events related to task $t$ are generated and added into the heap of events.

## 5.3 Algorithm

The *sweep_min* part of the synchronized sweep algorithm with precedence constraints consists again of a main loop, a processing events part and a filtering part. The processing part calls the algorithm $\mathrm{release\_task}$, which releases a task when the earliest starts of all its predecessors have been adjusted to their final values. We omit the filtering part since it is strictly identical to the one introduced in Section 4.

### 5.3.1 Main Loop

The main loop (Algorithm 7) consists of the following parts:

- `[CREATING EVENTS]` (line 2). The events are generated wrt. the start and end variables of each task that has no predecessors and inserted into $h\_events$.
- `[INITIALIZATION]` (lines 4 to 7). The available space $gap_r$ and the previous available space $gap'_r$ of each resource $r$ is set the corresponding resource limit $capa_r$. For each task $t$, its status is set to $none$ if $t$ is not fixed, $ready$ otherwise.
- `[MAIN LOOP]` (lines 9 to 14). For each sweep-line position the main loop processes all the corresponding events and updates the sweep-line status. Algorithm 7 returns **false** if a resource overflow occurs or if a task $t$ cannot be introduced in its temporal window because of its predecessors.

### 5.3.2 The Event Processing Part

In order to update the sweep-line status, Algorithm 8 reads and processes all the events related to the current sweep-line position $\delta$ and determines the sweep interval $[\delta, \delta_{next})$. Since this algorithm only differs from Algorithm 5 from line 22 to line 32, we do not comment again the other lines.

**ALGORITHM** sweep_min() : boolean

```
1:  [CREATING EVENTS]
2:  h_events ← generation of events wrt. n, s_t, s̄_t, d_t, e_t and the precedence constraints.
3:  [INITIALIZATION]
4:  for r = 0 to k − 1 do
5:     gap_r, gap'_r ← capa_r
6:  for t = 0 to n − 1 do
7:     if s_t = s̄_t then ring_t ← ready else ring_t ← none
8:  [MAIN LOOP]
9:  while ¬empty(h_events) do
10:    ⟨δ, δ_next, success⟩ ← process_events()
11:    if ¬success then
12:       return false
13:    if ¬filter_min(δ, δ_next) then
14:       return false
15: return true
```

**Algorithm 7:** Main sweep algorithm. Returns **false** if a resource overflow occurs or a precedence constraint cannot be satisfied, **true** otherwise. Ensures Property 3 in the latter case.

- `[PROCESSING RELEASE SUCCESSOR (RS) EVENTS]` (lines 23 to 32). When the sweep-line reaches the $RS$ event of a task $t$, we first have to determine whether or not it is its final position, i.e. whether the earliest end of task $t$ can still be updated. This requires the following steps to be considered:
  - If the status of task $t$ is set to $conflict$, the $RS$ event is pushed back to its first feasible position, i.e. $\delta + d_t$ (see line 25). This position considers that the earliest start of task $t$ will be adjusted to $\delta$.
  - Else if the position of the $RS$ event does not correspond to the earliest end of the task, meaning that the earliest end of task $t$ has been adjusted since the creation of the $RS$ event, we just push back the event to its correct position $e_t$ (see line 27).

  If the $RS$ event is at its final position, meaning that the earliest start of task $t$ will not be adjusted anymore, the successors of task $t$ are scanned. For each successors $t'$ of task $t$, the number of remaining tasks to filter wrt Property 3 (i.e. $nbpred_{t'}$) is decremented (see line 30). If the earliest starts of all predecessors of a task $t$ are updated wrt Property 3, i.e $nbpred_t = 0$, then the events related to task $t'$ are generated and inserted into the heap of events. This last part is described in Section 5.3.3, Algorithm 9.

### 5.3.3 Releasing a Successor

Once the earliest starts of all predecessors of a task $t$ have been adjusted wrt. Property 3 the algorithm release_task generates and adds the events of task $t$ into the heap of events or directly into the list of events that have just been extracted $\mathcal{E}$. This algorithm consists of three parts:

- `[CHECK THE NEW EARLIEST START]` (lines 2 to 3). This first part removes from the start (resp. end) variable of task $t$ all the values strictly less than $\delta$ (resp. strictly less than $\delta + d_t$). It returns **false** if one domain becomes empty.
- `[EARLIEST START OF TASK` $t$ `IS ADDED AT` $\delta$`]` (lines 5 to 16). First we consider the case where the earliest start of task $t$ is equal to $\delta$. If the start of task $t$ is fixed (i.e. $\underline{s_t} = \overline{s_t}$, line 6) then the available spaces are decreased wrt. the heights of task $t$. Otherwise a $PR$ event is added into the list of events to handle at the current sweep-line position (line 11). Since the $SCP$ and $ECPD$ events cannot be associated with the position $\delta$ of the sweep-line they are added into the heap of events. Finally, if task $t$ has a least one successor, a $RS$ event is generated.

- [EARLIEST START OF TASK $t$ IS ADDED AFTER $\delta$] (lines 18 to 27). We consider the case where the earliest start of task $t$ is strictly greater than $\delta$. In such a case, events are generated as in the initialization of *sweep_min* and added into the heap of events.

*Continuation of Example 3 (Illustrating the Synchronized Sweep Algorithm with Precedences).* The synchronized sweep algorithm with precedences first initializes the current sweep-line position to 0, $gap_0$ to 3, and $gap_1$ to 2. The status of task $t_0$ is set to $ready$, and to $none$ for all the other tasks. The sweep-line reads the $PR$ events of task $t_1$ and $t_2$ and sets their status to $check$ since their heights are less than the available spaces and their duration is greater than the sweep interval $[0, 1)$ (Algorithm 8, line 40). Nothing can be deduced by the first call to $\mathrm{filter\_min}$. Then it moves the sweep-line to position 1, reads the $SCP$ event of task $t_0$ and sets the available spaces $gap_0$ and $gap_1$ to 1. The call to $\mathrm{filter\_min}$ modifies the status of task $t_1$ to $conflict$, because of its height on resource $r_0$, and the status of task $t_2$, because of its height on resource $r_1$. Then it moves the sweep-line to position 2, reads the $ECPD$ event of task $t_0$ and sets the available spaces $gap_0$ to 3 and $gap_1$ to 2, and reads the three $RS$ events of tasks $t_0$, $t_1$ and $t_2$. Since task $t_0$ is initially fixed, its $RS$ event is well placed and we can scan its only successor, task $t_3$ (Algorithm 8, line 29). Consequently $nbpred_3$ is set to 1, meaning that exactly one predecessor of task $t_3$ is not yet adjusted to its fixpoint. Since the status of tasks $t_1$ and $t_2$ is set to $conflict$, their $RS$ event is pushed back to 4 (Algorithm 8, line 25). The call of $\mathrm{filter\_min}$ changes the status of tasks $t_1$ and $t_2$ to $check$ and adjusts their earliest start to 2. The event $\langle ECPD, t, 4\rangle$ is created for task $t_1$ reflecting the creation of a compulsory part. Then it moves the sweep-line to position 3, reads the $SCP$ event of task $t_1$ and sets $gap_0$ to 1 and $gap_1$ to 1. The status of the task $t_1$ is now set to $ready$ (Algorithm 8, line 9). The call of $\mathrm{filter\_min}$ modifies the status of task $t_2$ to $conflict$ because of its height on resource $r_1$. Then it moves the sweep-line to position 4, sets the available spaces $gap_0$ to 3 and $gap_1$ to 2 because of the end of compulsory part of task $t_1$. The sweep-line reads the $RS$ event of task $t_1$, which is now at its final position. So $nbpred_3$ is set to 0, meaning that all the predecessors of task $t_3$ have reached their fixpoint, and that the events of task $t_3$ can be generated and added into the heap of events. This part is handled by $\mathrm{release\_task}$ called on line 32 of Algorithm 8. In $\mathrm{release\_task}$, the earliest start of task $t_3$ is adjusted to $\delta$ (i.e. 4). Since task $t_3$ is not fixed and has no compulsory part, the following events are generated $\langle PR, t, 4\rangle$, $\langle SCP, t, 9\rangle$ (Algorithm 9, lines 11 to 12). Then, the $PR$ event is immediately processed and the status of task $t_3$ is set to $ready$ (Algorithm 8, line 42). The call of $\mathrm{filter\_min}$ sets the status of task $t_2$ to $check$, adjusts its earliest start to 4 and add the event $\langle ECPD, t, 6\rangle$, reflecting the creation of its compulsory part. Then it moves the sweep-line to position 5, reads the $SCP$ events of task $t_2$ and sets the available spaces $gap_0$ to 2 and $gap_1$ to 0. The status of task $t_2$ is now set to $check$ (Algorithm 8, line 9). Nothing can be deduced by $\mathrm{filter\_min}$. Then it moves the sweep-line to position 6, reads the $ECPD$ event of task $t_2$ and sets the available spaces $gap_0$ to 3 and $gap_1$ to 2. It also reads the $RS$ event of task $t_2$ which is at its final position. Consequently, $nbpred_4$ is set to 0 (Algorithm 8, line 30), meaning that all the earliest start of the predecessors of task $t_4$ are adjusted wrt Property 3. The call to $\mathrm{release\_task}$ generates the events $\langle PR, t, 6\rangle$, $\langle SCP, t, 7\rangle$ and $\langle ECPD, t, 9\rangle$. Finally, it successively moves the sweep-line to positions 7 and 9, corresponding to the start and end of the compulsory part of task $t_4$, and checks that the resource limits are never exceeded. □

## 5.4 Complexity

Given a *k-dimensional cumulative with precedences* involving $n$ tasks, the worst-case time complexity of the synchronized sweep algorithm with precedences is $O(kn^2 + nX(k + \log n))$, where $X$ is the maximum number of times that a $RS$ event can be shifted on the time axis. In the worst-case, for a task $t$, the $RS$ event can be pushed $\lceil(\overline{e_t} - \underline{e_t})/d_t\rceil$ times (see Algorithm 8, line 25). Over a full sweep, the worst-case time complexity of Algorithm 8 is $O(kn + n\log n + nX\log n)$. The part $nX\log n$ that is not present in the worst-case time complexity of the $k$-dimensional sweep without precedence is explained by the fact that we need to handle the $O(nX)$ $RS$ events. Over a full sweep, the worst-case time complexity of Algorithm 6 is $O(kn^2 + nXk)$. Due to the $O(nX)$ $RS$ events, Algorithm 6 can be called $O(nX)$ times with $\forall r \in [0..k-1] : gap_r = gap'_r$. In such a case, the complexity of Algorithm 6 is limited to $O(k)$ (see lines 6 and 12).

# 6 Synthesis

This section provides a synthetic view of the three sweep based filtering algorithms introduced in Sections 3, 4 and 5. First, we recall for each of them the key points concerning the events generated and processed, the information maintained by the sweep-line status and the worst-case time complexity. Second, we give the main principle of the greedy modes of these algorithm.

## 6.1 The Key Points of the New Sweep Algorithms

We begin with the 1-dimensional dynamic sweep introduced in Section 3 for the *cumulative* constraint:

- `[EVENTS]` It generates and inserts at most four events per task into the heap of events. When a task is initially not fixed (i.e. $\underline{s_t} \neq \overline{s_t}$), one $PR$ event related to its earliest start and one $CCP$ event related to its latest start are generated. Then, the $CCP$ event can be converted into a $SCP$ and an $ECPD$ event if a compulsory part occurs. The key events are the *conditional* $CCP$ and the *dynamic* $ECPD$ since they permit to handle the extension of the CPP in one single sweep.

- `[SWEEP-LINE STATUS]` The main data structures are the two heaps $h\_check$ and $h\_conflict$, which handle the status of the tasks. Indeed, the use of heaps is the key point to avoiding to systematically rescan all the active tasks each time the sweep-line moves.

- `[COMPLEXITY]` The worst-case time complexity of the 1-dimensional algorithm is $O(n^2 \log n)$. It can be reduced to $O(n^2)$ by replacing the two heaps $h\_check$ and $h\_conflict$ by a list that records the status of the tasks, but in such a case, the complexity $O(n^2)$ is more often reached in practice.

We continue with the synchronized sweep algorithm introduced in Section 4 for the *k-dimensional cumulative* constraint:

- `[EVENTS]` It generates and inserts at most four events per task into the heap of events. Compared to the 1-dimensional sweep, the $CCP$ has been merged with the $SCP$ event. Initially at most three events are generated per task, i.e. $PR$, $SCP$ and $ECPD$ events, then the $ECPD$ event can be pushed back on the time axis at most once. The key events in order to handle the extension of the CPP are the $SCP$ events that are generated for the tasks initially without compulsory part and the $ECPD$ events.

- `[SWEEP-LINE STATUS]` The main data structures are the circular double linked lists that record the status of the tasks. Unlike the 1-dimensional sweep, we don't use heaps to record the status of the tasks. Indeed, the advantage given by the heaps in the 1-dimensional sweep comes from the fact that an active task is either in the heap $h\_conflict$ or in the heap $h\_check$. For the $k$-dimensional version, we would have to create these two heaps for each resource, and a task would have to be duplicated in the heaps $h\_check$ to state that the task is not in conflict.

- `[COMPLEXITY]` The worst-case time complexity of the synchronized sweep algorithm is $O(kn^2)$.

We finish with the extension of the synchronized sweep algorithm introduced in Section 5 for the *k-dimensional cumulative with precedences* constraint:

- `[EVENTS]` Initially, it generates and inserts all the events that the synchronized sweep algorithm without precedences generates, plus one $RS$ event associated to the earliest end of the tasks that have at least one successor. In the worst-case, each $RS$ event related to a task $t$ can be pushed back on the time axis $\lceil (\overline{e_t} - \underline{e_t})/d_t \rceil$ times.

- `[SWEEP-LINE STATUS]` As for the synchronized sweep without precedences, the main data structures are the circular double linked lists that record the status of the tasks. To handle the precedences, we just add an integer $nbpred_t$ for each task $t$ that records the number of predecessors for which the final value of the earliest start was not yet found at the current sweep-line position.

- [COMPLEXITY] The worst-case time complexity of the synchronized sweep algorithm with precedences is $O(kn^2 + nX(k + \log n))$, where $X$ is the maximum number of times that a $RS$ event can be pushed back, i.e. $\max_{t \in [0..n-1]} (\lceil (\overline{e_t} - \underline{e_t})/d_t \rceil)$.

### 6.2 The Greedy Mode

The motivation for greedy modes is to handle larger instances in a CP solver. For each of the three sweep algorithms introduced in this report we design a greedy mode which reuses the *sweep_min* part of the corresponding filtering algorithm, in the sense that once the minimum value of a start variable is found, the greedy mode directly fixes the start to its earliest feasible value wrt Property 1, 2 or 3. Then the sweep-line is reset to this start and the process continues until all tasks get fixed or a resource overflow occurs. Thus the greedy modes directly benefit from the propagation performed while sweeping.

## 7 Evaluation

We implemented the dynamic sweep algorithm in Choco [22] and SICStus [23]. Choco benchmarks were run with an Intel Xeon at 2.93 GHz processor on one single core, memory limited to 14GB under Mac OS X 64 bits. SICStus benchmarks were run on a quad core 2.8 GHz Intel Core i7-860 machine with 8MB cache per core, running Ubuntu Linux (using only one processor core). The sweep algorithms that we consider in this section are:

| | |
|---|---|
| S | The 2001 sweep algorithm [16] |
| UH | The dynamic sweep algorithm, as described in Section 3 |
| UR | The dynamic sweep algorithm, but with the $ring$ data structure instead of heaps |
| K | The $k$-dimensional dynamic sweep algorithm, as described in Section 4 |
| KG | A greedy assignment algorithm corresponding to the previous item |
| P | The $k$-dimensional dynamic sweep algorithm with precedences described in Section 5 |
| PG | A greedy assignment algorithm corresponding to the previous item |

We have run our sweep algorithms with randomly generated instances, with resource-constrained project scheduling instances coming from PSPLib, and with randomized multi-year project scheduling instances coming from an industrial customer.

### 7.1 Random Instances

This experiment was run in Choco. The program listing of the instance generator is given in Appendix A. We ran random instances of bin-packing (unit duration) and cumulative (duration $\geq 1$) problems, with precedences or without them, with $k$ varying from 1 to 64 and $n$ from 1000 to 1024000. Instances were randomly generated with a density close to 0.7. For a given number of tasks, we generated two different instances with the average number of tasks overlapping a time point equal to 5. We measured the time needed to find a first solution. As a search heuristic, the variable with the smallest minimal value was chosen, and for that variable, the domain values were tried in increasing order. All instances were solved without backtracking. The times reported are total execution time, not just the time spent in the dynamic sweep algorithm.

In a first set of runs (see Figure 5), we compared algorithms UH, UR, K and KG on bin-packing instances without precedences. We note that UR is uniformly some 5% faster than UH, confirming the hypothesis that the rings data structure outperforms the heaps one. A preliminary analysis of the observed runtimes as a function of $n$ and $k$ suggest that UR solves instances in approx. $O(kn^{2.10})$ time, whereas K solves them in approx. $O(k^{0.25}n^{2.25})$ time. In other words, we observe a speed-up by nearly $k^{0.75}$. The

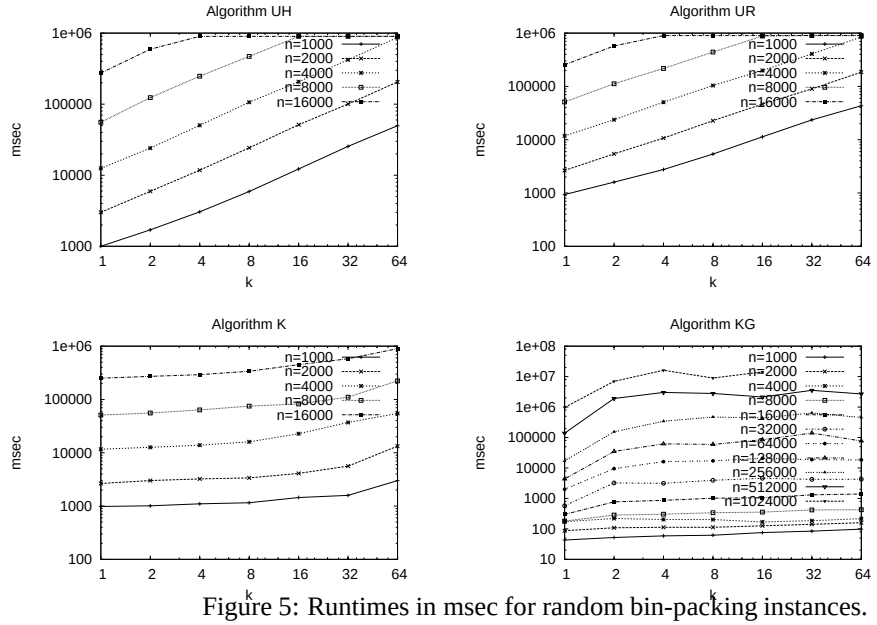

Figure 5: Runtimes in msec for random bin-packing instances.

pattern for KG is a little irregular, but we observe that the runtimes increase very little with increasing $k$, and also that the runtimes are orders of magnitude smaller than for K. KG is able to solve instances with more than one million tasks and 64 resources.

In a second set of runs (see Figure 6), we compared algorithms UH, UR, K, P and PG on bin-packing instances with precedences. We observe the same pattern for UH, UR and K as for the first set. Regarding K vs. P, P is uniformly some 15% to 50% faster than K, confirming the efficiency of treating cumulative and precedences globally. Regarding PG, curiously, the dependence of $k$ is similar to that of P, which was not the case for KG vs. K. Like for KG vs. K, the runtimes of PG are orders of magnitude smaller than for P.

In a third set of runs (see Figure 7), we compared algorithms UH, UR, K and KG on cumulative instances without precedences. In terms of the complexity analysis of runtimes as a function of $n$ and $k$, the picture is similar to that of the first set, but runtimes are about 50% longer.

In a fourth set of runs (see Figure 8), we compared algorithms UH, UR, K, P and PG on cumulative instances with precedences. In terms of the complexity analysis of runtimes as a function of $n$ and $k$, the picture is similar to that of the second set, but runtimes are about twice as long.

## 7.2 Resource-Constrained Project Scheduling

This experiment was run in SICStus Prolog. The program listing of the solver is given in Appendix B. We used single-mode resource-constrained project scheduling benchmark suites from PSPLib [1], comparing S, UH, K and P. There are four suites: J30, J60, J90 and J120. Each instance involves 30, 60, 90 or 120 tasks, respectively, 4 resources and several precedence constraints. The problem constraints were encoded as follows, depending on the algorithm used:

**S and UH** Four *cumulative* constraints, over the tasks using a nonzero amount of the given resource only, typically about 50% of all the tasks. Precedence constraints as simple linear inequalities over an end and a start variable.

**K** One $k$-dimensional *cumulative* constraint, over all the tasks. For tasks that did not use a given resource, a zero resource consumption was specified. Precedence constraints as above.

**P** As for algorithm K, but with precedences encoded as parameters to the *cumulative* constraint, instead of being posted separately.

[1] `http://129.187.106.231/psplib/`

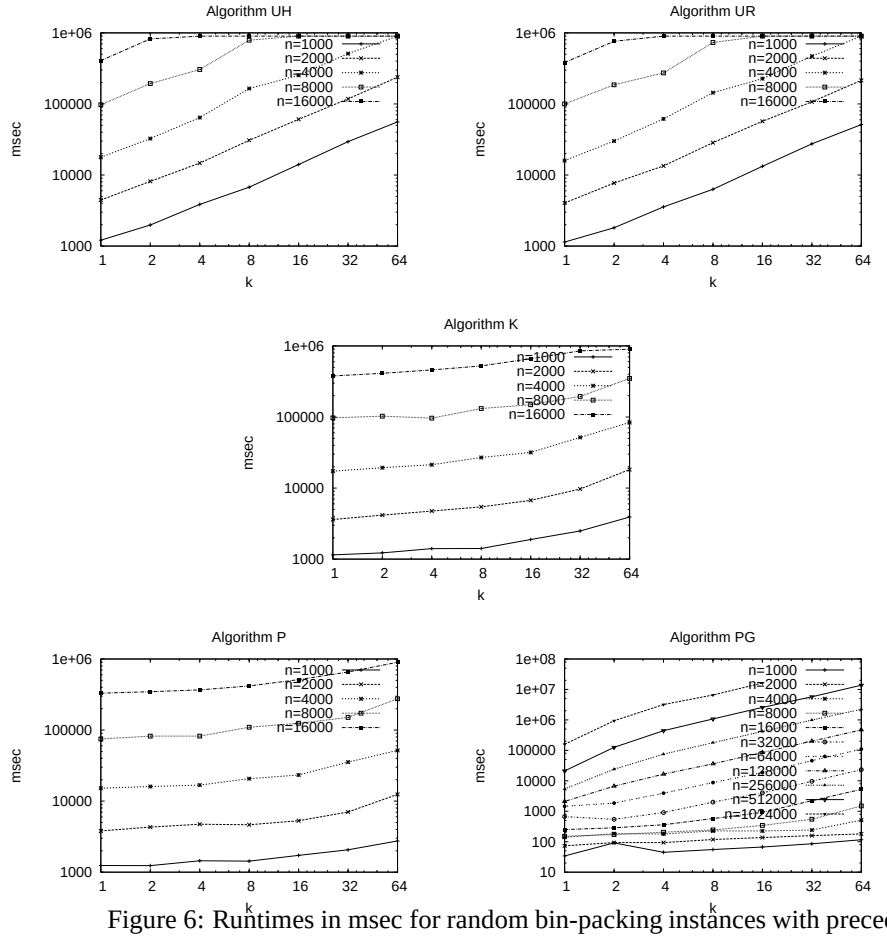


Figure 6: Runtimes in msec for random bin-packing instances with precedences.

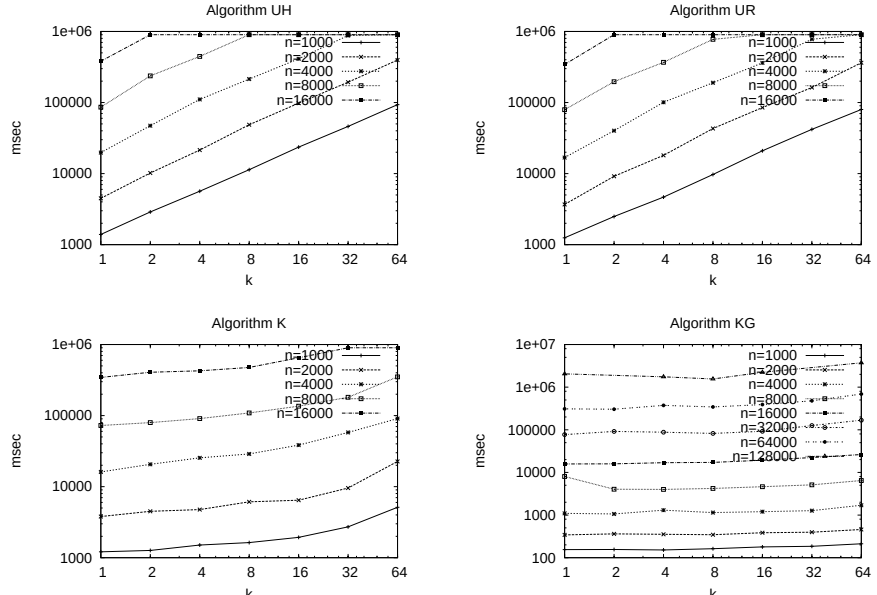


Figure 7: Runtimes in msec for random cumulative instances.

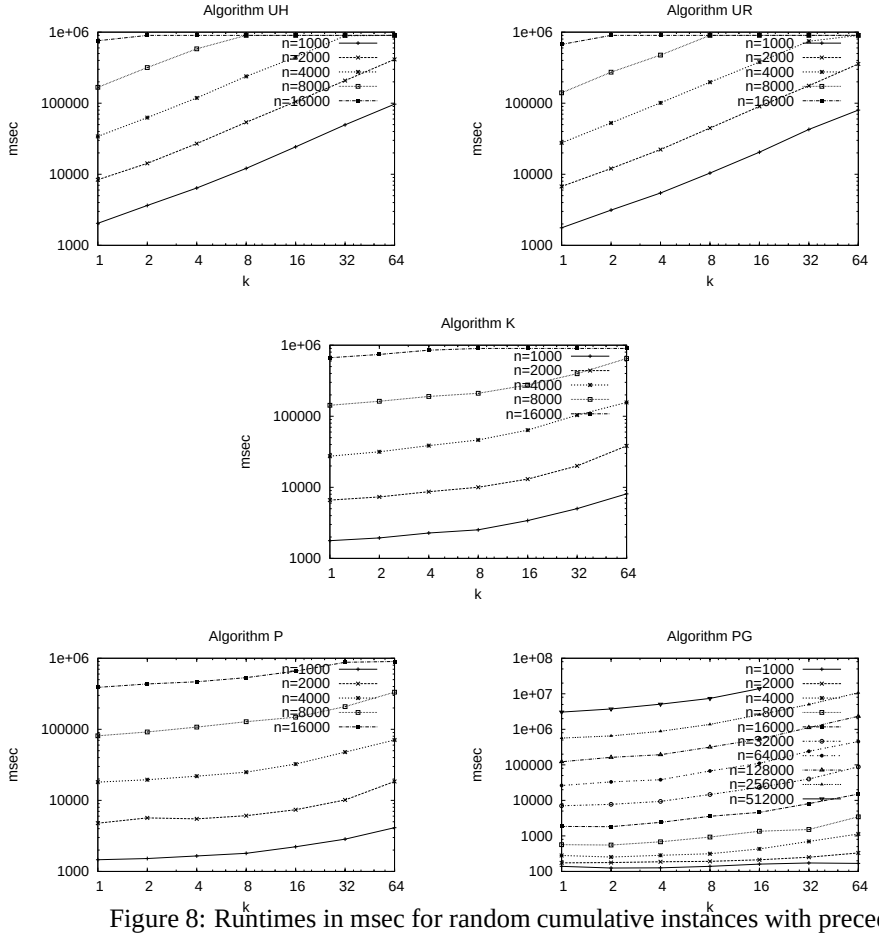


Figure 8: Runtimes in msec for random cumulative instances with precedences.

The initial domains of the start times corresponded to the optimal makespan, if it was known, or the best known upper bound, otherwise. A 60 seconds time limit per instance was imposed.

We used a two-phase search procedure. Phase one is nondeterministic, so if Phase two fails, it will backtrack into Phase one to find another partial solution, and so on:

**Phase one.** First, the tasks were statically ordered by descending area, where the area of a task is defined as its duration times its total resource consumption over the different resources. Then, for each task $i$ with start variable $s_i$ and duration $d_i$, we introduced variables $b_i$ and $u_i$ subject to $0 \leq u_i < d_i$ and $d_i b_i + u_i = s_i$. Finally, to ensure that each task has a compulsory part, we labeled the $b_i$ variables in the static order, by increasing value.

**Phase two.** Until all start variables have been fixed:

① Select the task $k$ with the smallest earliest start, breaking ties by choosing the earliest one in the static order.

② Split the current search tree node into a left node imposing $s_k \leq m$ and a right node imposing $s_k > m$, where $m = \lfloor (\underline{s_k} + \overline{s_k})/2 \rfloor$.

It is worth noting that the search tree for a given instance will be identical for all the algorithms, except S. Since algorithm S can filter out values in the middle of domains, it is able to solve some instances in slightly fewer backtracks than the other algorithms.

In Table 3, we show the results in terms of backtracks per second (bts) per suite and algorithm. Each table row corresponds to the set of observed bts for instances that took nonzero time and backtracks, showing the minimum, maximum, mean, median bts as well as the standard deviation and the number of instances solved in 60 seconds. Note that the reported bts numbers include both solved and timed out instances.

We observe from the **mean** and **median** columns that algorithm S is slower than UH, which is slower than K, which is slower than P, although for classes J90 and J120, there is practically no difference between K and P. Recall that the motivation for handling precedences directly in the filtering algorithm of P was to reach a fixpoint faster, requiring fewer invocations of the filtering algorithm than if the precedences are handled outside the algorithm. We conjecture that for J30 and J60, this is indeed what happens, whereas for J90 and J120, the saving is smaller and just about outweighs the overhead paid by P for handling precedences, what with extra dynamic events and everything.

In Table 4, we give for each suite a pairwise comparison of the algorithms. Each table row corresponds to the set of observed (bts for algorithm $x$)/(bts for algorithm $y$) for given algorithms $x/y$ and instances that took nonzero time and backtracks to solve for both algorithms.

The latter table confirms the findings of the former one, and show that the largest performance gain in our series of algorithm is due to the handling of $k$ resources in one constraint. The UH/K quotients are slightly larger than $4^{-0.75} = 0.35$, predicted in the analysis of Figure 5. We conjecture that this is due to the abundance of tasks with zero demand for one or more resources in the PSPLib instances, which means that each individual 1-dimensional constraint needs to deal only with a subset of the tasks.

## 7.3 An Industrial Application

This experiment was run in SICStus Prolog, except algorithm PG, which was run in Choco. It consists of a resource-constrained project scheduling problem [24] with 8 resources and up to 15000 tasks. The resource usage array is sparse: only 12.5% of the array elements are nonzero.

The data are a randomized example of a multi-year project scheduling problem from industrial customer. A series of jobs have to be scheduled over multiple years, each job consisting of multiple tasks, which may need some of the limited resources. Links between tasks of different jobs indicate dependencies in the workflow.

The key point is that we are not solving the problem once, to come up with an operational plan, but have to solve many what-if scenarios, where the user changes the timing of the migration tasks, the mix

Table 3: Results for PSPLib instances per suite and algorithm (runtime in msec).

| class | #instances | algorithm | #solved | min | max | mean | median | stddev |
|---|---|---|---|---|---|---|---|---|
| J30 | 480 | S | 469 | 100.00 | 34040.00 | 3147.90 | 2827.78 | 3052.43 |
| | | UH | 471 | 100.00 | 56733.33 | 4034.65 | 3200.00 | 5077.59 |
| | | K | 474 | 1300.00 | 34040.00 | 9332.15 | 8212.50 | 5087.08 |
| | | P | 475 | 100.00 | 34500.00 | 12437.60 | 11635.82 | 5806.21 |
| J60 | 480 | S | 368 | 50.00 | 20600.00 | 1748.56 | 1533.91 | 1748.79 |
| | | UH | 369 | 100.00 | 17430.77 | 2165.60 | 1914.70 | 1807.89 |
| | | K | 374 | 100.00 | 14700.00 | 5650.81 | 5539.14 | 2859.14 |
| | | P | 374 | 100.00 | 10622.64 | 5915.70 | 6503.00 | 2603.33 |
| J90 | 480 | S | 293 | 50.00 | 18890.94 | 1457.08 | 994.43 | 1758.14 |
| | | UH | 294 | 33.33 | 23923.77 | 2106.73 | 1437.25 | 2611.13 |
| | | K | 296 | 50.00 | 9510.00 | 4184.08 | 4489.25 | 2298.52 |
| | | P | 295 | 50.00 | 10568.43 | 4138.43 | 4648.68 | 2193.36 |
| J120 | 600 | S | 91 | 50.00 | 12779.59 | 1617.27 | 877.87 | 1873.42 |
| | | UH | 93 | 50.00 | 29948.11 | 3519.59 | 2117.19 | 4138.10 |
| | | K | 95 | 33.33 | 12450.80 | 5239.94 | 5611.33 | 2016.51 |
| | | P | 94 | 50.00 | 9455.08 | 5283.45 | 5681.04 | 1764.89 |

Table 4: Results for PSPLib instances per suite and pair of algorithms (runtime in msec).

| class | #instances | algorithms | min | max | mean | median | stddev |
|---|---|---|---|---|---|---|---|
| J30 | 480 | S/UH | 0.25 | 2.00 | 0.91 | 0.91 | 0.28 |
| | | S/K | 0.12 | 2.00 | 0.45 | 0.36 | 0.29 |
| | | S/P | 0.08 | 1.80 | 0.32 | 0.28 | 0.23 |
| | | UH/K | 0.14 | 2.00 | 0.51 | 0.42 | 0.31 |
| | | UH/P | 0.09 | 3.00 | 0.39 | 0.30 | 0.35 |
| | | K/P | 0.25 | 2.00 | 0.88 | 0.75 | 0.48 |
| J60 | 480 | S/UH | 0.34 | 2.50 | 0.89 | 0.86 | 0.34 |
| | | S/K | 0.10 | 2.55 | 0.40 | 0.32 | 0.29 |
| | | S/P | 0.08 | 3.18 | 0.38 | 0.30 | 0.33 |
| | | UH/K | 0.13 | 2.15 | 0.48 | 0.38 | 0.30 |
| | | UH/P | 0.14 | 2.69 | 0.44 | 0.35 | 0.31 |
| | | K/P | 0.39 | 2.00 | 0.96 | 0.95 | 0.30 |
| J90 | 480 | S/UH | 0.25 | 3.00 | 0.78 | 0.70 | 0.36 |
| | | S/K | 0.06 | 3.00 | 0.44 | 0.28 | 0.42 |
| | | S/P | 0.06 | 2.47 | 0.43 | 0.28 | 0.40 |
| | | UH/K | 0.12 | 4.00 | 0.57 | 0.38 | 0.56 |
| | | UH/P | 0.12 | 4.00 | 0.57 | 0.37 | 0.54 |
| | | K/P | 0.44 | 2.00 | 1.02 | 1.00 | 0.30 |
| J120 | 600 | S/UH | 0.22 | 2.00 | 0.49 | 0.40 | 0.30 |
| | | S/K | 0.05 | 3.00 | 0.36 | 0.18 | 0.41 |
| | | S/P | 0.05 | 3.00 | 0.37 | 0.18 | 0.47 |
| | | UH/K | 0.16 | 3.33 | 0.71 | 0.46 | 0.65 |
| | | UH/P | 0.16 | 3.87 | 0.72 | 0.42 | 0.71 |
| | | K/P | 0.33 | 3.00 | 1.01 | 1.00 | 0.23 |

of resource limits, etc. This means that fast, interactive response is very important, and consequently the availability of a greedy method that can handle several *cumulative* and precedence constraints is crucial.

We compared algorithms S, UH, UR, K, P and PG on these instances, as shown in Table 5, displaying runtimes and numbers of invocations of the filtering algorithm. The instances are easy, and are solved without backtracking by all the algorithms. The same search strategy was used as in Section 7.1.

We find that S is slower than all the algorithms introduced in this report. UH was slower than UR, confirming the earlier finding that the rings data structure outperforms the heaps one. UR was faster than K, which we conjecture is due to the sparse usage array. This also fits the observed number of invocations of the filtering algorithms. With all array elements nonzero, we would have expected about $k\ (= 8)$ times more invocations for UR than for K. Finally, the relatively poor performance of P vs. K can also be explained by the invocation counts. The small saving in number of invocations we see here clearly does not outweigh the extra overhead in P of handling precedences. We conjecture that with a harder instance and different search strategy, the difference in number of invocations would be greater.

Finally, this application with its large but easy instances and its requirement on speed and interaction demonstrates the usefulness of a greedy assignment mode.

# 8 Conclusion

Unlike the traditional way of propagating constraints where each constraint is propagated independently from each other, this paper exploits the idea of *synchronizing the propagation of different constraints* for getting more scalable scheduling constraints. Starting from one single *cumulative* constraint, we then consider several *cumulative* constraints and finally several *cumulative* and *precedence* constraints. The idea is not to use a sophisticated filtering algorithm that performs more deduction by considering a conjunction of constraints globally, but rather to perform some standard propagation in a faster way so that the filtering algorithm scales better as the number of tasks of a scheduling problem increases. All three algorithms introduced in this paper can operate both in filtering mode as well as in greedy assignment mode. Our benchmarks show that the filtering mode achieves a significant speed-up over the decomposition into independent *cumulative* and *precedence* constraints, especially as the number of *cumulative* or *precedence* constraints increases. The greedy mode yields another two orders of magnitude of speed up allowing an industrial problem of significant size to be solved in real time.

Table 5: Results for the industrial application. Runtimes in seconds. All instances were solved in less than two seconds by PG.

| instance | #tasks | #precedences | algorithm | runtime | #invocations |
|---|---|---|---|---|---|
| A | 8268 | 31538 | S | 46.01 | 8471 |
| | | | UH | 20.82 | 8300 |
| | | | UR | 14.97 | 8303 |
| | | | K | 24.23 | 8308 |
| | | | P | 30.00 | 8269 |
| B | 7628 | 26711 | S | 25.96 | 7794 |
| | | | UH | 14.54 | 7652 |
| | | | UR | 10.49 | 7655 |
| | | | K | 14.15 | 7660 |
| | | | P | 23.91 | 7629 |
| C | 7467 | 27055 | S | 24.36 | 7631 |
| | | | UH | 13.79 | 7493 |
| | | | UR | 9.79 | 7496 |
| | | | K | 13.47 | 7501 |
| | | | P | 22.63 | 7468 |
| D | 8024 | 28017 | S | 30.88 | 8196 |
| | | | UH | 16.45 | 8051 |
| | | | UR | 11.85 | 8054 |
| | | | K | 16.28 | 8059 |
| | | | P | 26.66 | 8025 |
| E | 6421 | 22895 | S | 15.41 | 6557 |
| | | | UH | 9.32 | 6440 |
| | | | UR | 6.54 | 6443 |
| | | | K | 9.55 | 6448 |
| | | | P | 16.35 | 6422 |
| F | 6347 | 22943 | S | 14.21 | 6459 |
| | | | UH | 9.00 | 6362 |
| | | | UR | 6.30 | 6365 |
| | | | K | 9.44 | 6370 |
| | | | P | 16.08 | 6348 |
| G | 14337 | 53218 | S | 115.77 | 14734 |
| | | | UH | 62.75 | 14403 |
| | | | UR | 51.65 | 14406 |
| | | | K | 57.90 | 14411 |
| | | | P | 99.83 | 14338 |
| H | 11354 | 41776 | S | 73.91 | 11638 |
| | | | UH | 37.56 | 11402 |
| | | | UR | 28.82 | 11405 |
| | | | K | 36.28 | 11410 |
| | | | P | 59.61 | 11355 |
| I | 13348 | 50311 | S | 105.46 | 13669 |
| | | | UH | 54.38 | 13405 |
| | | | UR | 42.76 | 13408 |
| | | | K | 50.45 | 13413 |
| | | | P | 85.77 | 13349 |
| J | 15351 | 59917 | S | 131.91 | 15746 |
| | | | UH | 73.52 | 15422 |
| | | | UR | 54.75 | 15425 |
| | | | K | 76.49 | 15430 |
| | | | P | 115.94 | 15352 |
| K | 14945 | 62541 | S | 121.25 | 15318 |
| | | | UH | 67.37 | 15008 |
| | | | UR | 51.53 | 15011 |
| | | | K | 69.00 | 15016 |
| | | | P | 113.85 | 14946 |

**ALGORITHM** process_events() : $\langle \text{integer}, \text{integer} \rangle$

1: $\langle \delta, \mathcal{E} \rangle \leftarrow$ extract and record in $\mathcal{E}$ all the events in $h\_events$ related to the minimal date $\delta$
2: `[PROCESSING START COMPULSORY PART (SCP) EVENTS]`
3: **for all** events of type $\langle SCP, t, \overline{s_t} \rangle$ in $\mathcal{E}$ **do**
4: $\quad ecp' \leftarrow \underline{e_t}$
5: $\quad$ **if** $ring_t = \text{conflict}_\star$ **then**
6: $\quad\quad \text{adjust\_min\_var}(s_t, \overline{s_t})$; $\text{adjust\_min\_var}(e_t, \overline{e_t})$
7: $\quad\quad ring_t \leftarrow \text{ready}$
8: $\quad$ **else if** $ring_t = \text{check}$ **then**
9: $\quad\quad ring_t \leftarrow \text{ready}$
10: $\quad$ **if** $\delta < \underline{e_t}$ **then**
11: $\quad\quad$ **for** $r = 0$ **to** $k - 1$ **do**
12: $\quad\quad\quad gap_r \leftarrow gap_r - h_{t,r}$
13: $\quad\quad$ **if** $ecp' \leq \delta$ **then** // introduce $ECPD$ event if new compulsory part
14: $\quad\quad\quad$ add $\langle ECPD, t, \underline{e_t} \rangle$ to $h\_events$
15: `[PROCESSING DYNAMIC (ECPD) EVENTS]`
16: **for all** events of type $\langle ECPD, t, \underline{e_t} \rangle$ in $\mathcal{E}$ **do**
17: $\quad$ **if** $\underline{e_t} > \delta$ **then** // reintroduce $ECP$ event if $\underline{e_t}$ has moved
18: $\quad\quad$ add $\langle ECPD, t, \underline{e_t} \rangle$ to $h\_events$
19: $\quad$ **else**
20: $\quad\quad$ **for** $r = 0$ **to** $k - 1$ **do**
21: $\quad\quad\quad gap_r \leftarrow gap_r + h_{t,r}$
22: `[PROCESSING RELEASE SUCCESSOR (RS) EVENTS]`
23: **for all** events of type $\langle RS, t, \underline{e_t} \rangle$ in $\mathcal{E}$ **do**
24: $\quad$ **if** $ring_t = \text{conflict}_\star$ **then**
25: $\quad\quad$ add $\langle RS, t, \delta + d_t \rangle$ to $h\_events$ // push back the $RS$ event
26: $\quad$ **else if** $\delta \neq \underline{e_t}$ **then**
27: $\quad\quad$ add $\langle RS, t, \underline{e_t} \rangle$ to $h\_events$ // push back the $RS$ event
28: $\quad$ **else**
29: $\quad\quad$ **for all** $t' \in successors_t$ **do** // scan the successors of task $t$
30: $\quad\quad\quad nbpred_{t'} \leftarrow nbpred_{t'} - 1$
31: $\quad\quad\quad$ **if** $nbpred_{t'} = 0$ **then**
32: $\quad\quad\quad\quad$ **if** $\neg\text{release\_task}(t', \delta, \mathcal{E})$ **then return false** // introduce events related to task $t'$
33: `[DETERMINE THE NEXT EVENT DATE]`
34: $\delta_{next} \leftarrow \text{get\_top\_key}(h\_events)$ // $+\infty$ if empty
35: `[PROCESSING EARLIEST START (PR) EVENTS]`
36: **for all** events of type $\langle PR, t, \underline{s_t} \rangle$ in $\mathcal{E}$ **do** // $PR$ must be handled last
37: $\quad$ **if** $\exists r \mid h_{t,r} > gap_r$ **then** // is task $t$ in conflict?
38: $\quad\quad ring_t \leftarrow \text{conflict}_r$
39: $\quad$ **else if** $\underline{e_t} > \delta_{next}$ **then** // might task be in conflict next time ?
40: $\quad\quad ring_t \leftarrow \text{check}$
41: $\quad$ **else**
42: $\quad\quad ring_t \leftarrow \text{ready}$
43: **return** $\langle \delta, \delta_{next} \rangle$

**Algorithm 8:** Called every time the sweep-line moves. Extracts and processes all events at given time point $\delta$. Returns the current $\delta$ and the next time point $\delta_{next}$ and a Boolean indicating whether the algorithm succeeds or not.

**ALGORITHM** release_task$(t, \delta, \mathcal{E})$ : boolean

1: [CHECK THE NEW EARLIEST START]
2: **if** $\neg$adjust_min_var$(s_t, \delta) \vee \neg$adjust_min_var$(e_t, \delta + d_t)$ **then**
3: **return false**
4: [EARLIEST START OF TASK $t$ IS ADDED AT $\delta$]
5: **if** $\underline{s_t} = \delta$ **then**
6: **if** $\underline{s_t} = \overline{s_t}$ **then** // task $t$ is scheduled and starts at $\delta$
7: **for** $r = 0$ **to** $k - 1$ **do**
8: $gap_r \leftarrow gap_r - h_{t,r}$
9: $ring_t \leftarrow$ ready
10: **else**
11: add $\langle PR, t, \underline{s_t} \rangle$ to $\mathcal{E}$ // add $PR$ event to $\mathcal{E}$ since it needs to be handled now
12: add $\langle SCP, t, \overline{s_t} \rangle$ to $h\_events$
13: **if** $\overline{s_t} < \underline{e_t}$ **then** // $ECPD$ event implies presence of compulsory part
14: add $\langle \overline{ECPD}, t, \underline{e_t} \rangle$ to $h\_events$
15: **if** task $t$ has a least one successor **then**
16: add $\langle RS, t, \underline{e_t} \rangle$ to $h\_events$
17: [EARLIEST START OF TASK $t$ IS ADDED AFTER $\delta$]
18: **else**
19: add $\langle SCP, t, \overline{s_t} \rangle$ to $h\_events$
20: **if** $\overline{s_t} < \underline{e_t}$ **then** // $ECPD$ event implies presence of compulsory part
21: add $\langle \overline{ECPD}, t, \underline{e_t} \rangle$ to $h\_events$
22: **if** $\underline{s_t} < \overline{s_t}$ **then** // task $t$ is not yet fixed
23: add $\langle PR, t, \underline{s_t} \rangle$ to $h\_events$
24: **else**
25: $ring_t \leftarrow$ ready
26: **if** task $t$ has at least one successor **then**
27: add $\langle RS, t, \underline{e_t} \rangle$ to $h\_events$
28: **return true**

**Algorithm 9:** Generates and adds events related to task $t$, meaning that all its predecessors have reached their fixpoint. Returns **false** for failure if $\delta$ has passed the latest start of task $t$, **true** otherwise.

# References


[1] E. Freuder, J. Lee, B. O'Sullivan, G. Pesant, F. Rossi, M. Sellman, and T. Walsh. The future of CP. personal communication, 2011.

[2] Barry O'Sullivan. CP panel position - the future of CP. personal communication, 2011.

[3] Jean-Charles Régin and Mohamed Rezgui. Discussion about constraint programming bin packing models. In *AI for Data Center Management and Cloud Computing*. AAAI, 2011.

[4] ROADEF. Challenge 2012 machine reassignment, 2012.

[5] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *CP'98*, volume 1520 of *LNCS*, pages 417–431. Springer, 1998.

[6] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Why cumulative decomposition is not as bad as it sounds. In *CP'09*, volume 5547 of *LNCS*, pages 746–761. Springer, 2009.

[7] Nicolas Beldiceanu and Mats Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In *CP'01*, volume 2237 of *LNCS*, pages 377–391. Springer, 2001.

[8] Roger Kameugne, Laure Pauline Fotso, Joseph Scott, and Youcheu Ngo-Kateu. A quadratic edge-finding filtering algorithm for cumulative resource constraints. In *CP'11*, volume 6876 of *LNCS*, pages 478–492. Springer, 2011.

[9] Petr Vilím. Edge finding filtering algorithm for discrete cumulative resources in $O(kn \log n)$. In *CP'09*, volume 5547 of *LNCS*, pages 802–816. Springer, 2009.

[10] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. International Series in Operations Research and Management Science. Kluwer, 2001.

[11] Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In *CPAIOR*, volume 6697 of *LNCS*, pages 230–245. Springer, 2011.

[12] N. Beldiceanu, M. Carlsson, E. Poder, R. Sadek, and C. Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic $k$-dimensional objects. In *CP'07*, volume 4741 of *LNCS*, pages 180–194. Springer, 2007.

[13] Christian Schulte. Comparing trailing and copying for constraint programming. In Danny De Schreye, editor, *ICLP'99*, pages 275–289. The MIT Press, 1999.

[14] Nicolas Beldiceanu, Mats Carlsson, and Sven Thiel. Sweep synchronisation as a global propagation mechanism. *Computers and Operations Research*, 33(10):2835–2851, 2006.

[15] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7):57–73, 1993.

[16] Nicolas Beldiceanu and Mats Carlsson. A new multi-resource *cumulatives* constraint with negative heights. In *CP 2002*, volume 2470 of *LNCS*, pages 63–79. Springer, 2002.

[17] Arnaud Letort, Nicolas Beldiceanu, and Mats Carlsson. A scalable sweep algorithm for the cumulative constraint. In *CP*, LNCS, pages 439–454. Springer, 2012.

[18] Arnaud Letort, Mats Carlsson, and Nicolas Beldiceanu. A synchronized sweep algorithm for the k-dimensional cumulative constraint. In *CPAIOR*, LNCS, pages ?–? Springer, 2013.

[19] Rainer Kolisch and Arno Sprecher. PSPLIB – a project scheduling problem library. *European Journal Of Operational Research*, 96:205–216, 1996.

[20] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational geometry - algorithms and Applications*. Springer, 1997.

[21] C. Le Pape. *Des systèmes d'ordonnancement flexibles et opportunistes*. PhD thesis, Université Paris IX, 1988. in French.

[22] CHOCO Team. Choco: an open source Java CP library. Research report 10-02-INFO, Ecole des Mines de Nantes, 2010.

[23] Mats Carlsson and et al. *SICStus Prolog User's Manual*. SICS, 4.2.3 edition, 2012.

[24] Helmut Simonis. An industrial benchmark. personal communication, 2013.

# A Source Code for Random Instance Generator

```
public class Generation {
    public static void main(String[] args) {
        // define parameters
        int nbTasks = 100;
        int nbResources = 3;
        double density = 0.8;
        int capacity = 10;
        int minHeight = 1;
        int maxHeight = 5;
        int minDuration = 1;
        int maxDuration = 10;
        double avgNbSuccessors = 3;
        int maxNbSuccessors = 9;
        // generate the instance
        RCPSPGenerator g = new RCPSPGenerator(nbTasks, nbResources, density,
            capacity, minHeight, maxHeight, minDuration, maxDuration,
            avgNbSuccessors, maxNbSuccessors);
        RCPSPInstance i = g.generateCumulative();
    }
}

import java.util.Random;

public class RCPSPGenerator {

    private final int nbTasks;
    private final int nbResources;
    private final double density;
    private final int capacity;
    private final int minHeight;
    private final int maxHeight;
    private final int minDuration;
    private final int maxDuration;
    private final double avgNbSuccessors;
    private final int maxNbSuccessors;
    private final Random rnd;
    private double avgTaskEnergy;
    private int makespan;

    public RCPSPGenerator(int nbTasks, int nbResources, double density, int
        capacity, int minHeight, int maxHeight,
                          int minDuration, int maxDuration, double
                              avgNbSuccessors, int maxNbSuccessors) {
        this.rnd = new Random();
        this.nbTasks = nbTasks;
        this.nbResources = nbResources;
        this.density = density;
        this.capacity = capacity;
        this.minHeight = minHeight;
        this.maxHeight = maxHeight;
        this.minDuration = minDuration;
        this.maxDuration = maxDuration;
        this.avgNbSuccessors = avgNbSuccessors;
        this.maxNbSuccessors = maxNbSuccessors;
    }
```

```
public RCPSPGenerator(int nbTasks, int nbResources, double density, int
    capacity, int minHeight, int maxHeight,
                      int minDuration, int maxDuration) {
    this(nbTasks,nbResources,density,capacity,minHeight,maxHeight,
        minDuration,maxDuration,0,0);
}

public RCPSPInstance generateCumulative() {
    // compute makespan
    this.avgTaskEnergy = ((maxDuration+minDuration)/2)*((minHeight+
        maxHeight)/2);
    double sumEnergy = (long) ((avgTaskEnergy*nbTasks)/density);
    this.makespan = (int) (sumEnergy / capacity);
    // memory alloc
    int[] startLB = new int[nbTasks];
    int[] duration = new int[nbTasks];
    int[] endUB = new int[nbTasks];
    int[][] heights = new int[nbTasks][nbResources];
    int[][] successors = new int[nbTasks][];
    // generate the duration and the height of the tasks for 1 resource
    int curNbTasks = 0;
    double futurEnergy = 0, avgFuturEnergy, taskEnergy, curEnergy = 0;
    int _d=-1, _h=-1;
    boolean isOk;
    while ( curNbTasks < nbTasks ) {
        avgFuturEnergy = (curNbTasks+1)*avgTaskEnergy;
        isOk = false;
        while (!isOk) {
            _d = random(minDuration,maxDuration);
            _h = random(minHeight,maxHeight);
            taskEnergy = _d * _h;
            futurEnergy = curEnergy + taskEnergy;
            if ( (futurEnergy <= avgFuturEnergy*1.02) && (futurEnergy >=
                avgFuturEnergy*0.08) ) {
                isOk = true;
            }
        }
        curEnergy = futurEnergy;
        startLB[curNbTasks] = 0;
        duration[curNbTasks] = _d;
        endUB[curNbTasks] = makespan;
        heights[curNbTasks][0] = _h;
        curNbTasks++;
    }
    // generate the heights of the tasks for the other dimensions
    for (int r=1;r<nbResources;r++) {
        curNbTasks = 0;
        curEnergy = 0;
        while ( curNbTasks < nbTasks ) {
            avgFuturEnergy = (curNbTasks+1)*avgTaskEnergy;
            isOk = false;
            while (!isOk) {
                _h = random(minHeight,maxHeight);
                taskEnergy = duration[curNbTasks] * _h;
                futurEnergy = curEnergy + taskEnergy;
                if ( (futurEnergy <= avgFuturEnergy*1.02) && (futurEnergy
                    >= avgFuturEnergy*0.08) ) {
                    isOk = true;
```

```
                }
            }
            curEnergy = futurEnergy;
            heights[curNbTasks][r] = _h;
            curNbTasks++;
        }
    }
    // generate the precedence relations (without cycle)
    if (avgNbSuccessors == 0 || maxNbSuccessors == 0) {
        for (int t=0;t<nbTasks;t++) {
            successors[t] = new int[0];
        }
    } else {
        int[] succTmp = new int[maxNbSuccessors];
        int nbSucc, currentWS;
        final int windowSize = Math.max((int)0.05*nbTasks,maxNbSuccessors)
            ;
        double percToBeSucc;
        for (int i=0;i<nbTasks;i++) {
            if ( i+windowSize<nbTasks ) {
                currentWS = windowSize;
            } else {
                currentWS = nbTasks - 1 - i;
            }
            nbSucc = 0;
            percToBeSucc = avgNbSuccessors / currentWS;
            for (int j=i+1;j<i+currentWS;j++) {
                if ( randomDouble() < percToBeSucc ) {
                    succTmp[nbSucc] = j;
                    nbSucc++;
                    if (nbSucc == maxNbSuccessors) {break;}
                }
            }
            successors[i] = new int[nbSucc];
            for (int j=0;j<nbSucc;j++) {
                successors[i][j] = succTmp[j];
            }
        }
    }
    // create a new instance
    RCPSPInstance instance = new RCPSPInstance();
    instance.nbTasks = nbTasks;
    instance.nbResources = nbResources;
    instance.startLB = startLB;
    instance.endUB = endUB;
    instance.duration = duration;
    instance.heights = heights;
    instance.capacities = new int[nbResources];
    for (int r=0;r<nbResources;r++) {
        instance.capacities[r] = capacity;
    }
    instance.successors = successors;
    return instance;
}

public void setSeed(long seed) {
    this.rnd.setSeed(seed);
}
```

```
    private int random(int lb, int ub) {
        return this.rnd.nextInt(ub-lb)+lb;
    }

    private double randomDouble() {
        return this.rnd.nextDouble();
    }
}

public class RCPSPInstance {

    public int[] startLB;
    public int[] endUB;
    public int[] duration;
    public int[][] heights;
    public int[][] successors;
    public int[] capacities;
    public int nbTasks;
    public int nbResources;

    RCPSPInstance() {}
}
```

# B Source Code for PSPLIB Instance Solver

```
:- use_module(library(lists)).
:- use_module(library(ugraphs)).
:- use_module(library(timeout)).
:- use_module(library(file_systems)).
:- use_module(library(clpfd)).
:- ensure_loaded(bounds).

top :-
        solve_dir(j120, static),
        solve_dir(j120, uni),
        solve_dir(j120, decomposed),
        solve_dir(j120, multi),
        solve_dir(j120, multi_precedences),
        solve_dir(j90, static),
        solve_dir(j90, uni),
        solve_dir(j90, decomposed),
        solve_dir(j90, multi),
        solve_dir(j90, multi_precedences),
        solve_dir(j60, static),
        solve_dir(j60, uni),
        solve_dir(j60, decomposed),
        solve_dir(j60, multi),
        solve_dir(j60, multi_precedences),
        solve_dir(j30, static),
        solve_dir(j30, uni),
        solve_dir(j30, decomposed),
        solve_dir(j30, multi),
        solve_dir(j30, multi_precedences),
        true.

solve_dir(Dir, Algo) :-
        solve_dir(Dir, Algo, mats_2phase).

solve_dir(Dir, Algo, Sel) :-
        atom_concat('../PSPLIB/', Dir, AbsDir),
        file_members_of_directory(AbsDir, Members),
        (   foreach(Relative-Absolute,Members),
            param(Algo,Sel,Dir)
        do  \+ \+ solve(Dir, Absolute, Relative, Algo, Sel)
        ).

solve(Dir, Abs, Rel, Algo, Sel) :-
        generate(Dir, Abs, Rel, Algo, Ss, Durs, Es, Hss, Ps,
                 Tasks1-Lim1, Tasks2-Lim2, Tasks3-Lim3, Tasks4-Lim4),
        statistics(runtime, _),
        fd_statistics(backtracks, _),
        fd_statistics(resumptions, _),
        disjunctives(Tasks1, Lim1),
        disjunctives(Tasks2, Lim2),
        disjunctives(Tasks3, Lim3),
        disjunctives(Tasks4, Lim4),
        post(Algo, [Tasks1-Lim1,Tasks2-Lim2,Tasks3-Lim3,Tasks4-Lim4], Hss, Ps,
             F),
        time_out(search(Sel,Ss,Durs,Es,Hss,F), 60000, Res),
        statistics(runtime, [_,T2]),
        fd_statistics(backtracks, Btr),
```

```
        fd_statistics(resumptions, Ru),
        atom_concat(Inst, '.sm', Rel),
        portray_clause(data(Inst,Algo,Res,T2,Btr,Ru)).

search(mats_2phase, Ss, Durs, _Es, Hss, _) :-
        (   foreach(V,Ss),
            foreach(C,Bins),
            foreach(D,Durs),
            foreach(Hs,Hss),
            foreach(V-Rank,Pairs),
            foreach(Rank-B,KL1),
            foreach(_-C,KL2)
        do  B #= V/D,
            sumlist(Hs, Hsum),
            Rank is -D*Hsum
        ),
        keysort(KL1, KL2),
        labeling([], Bins),
        mats_labeling(Pairs).

mats_labeling([]) :- !.
mats_labeling(Pairs0) :-
        (   foreach(Pair, Pairs0),
            fromto(Pairs,Pairs1,Pairs2,[]),
            fromto(([[none]]-0)-0,Key1,Key2,_-O2)
        do  Pair = O-R,
            (   nonvar(O) ->
                Pairs1 = Pairs2,
                Key1 = Key2
            ;   fd_set(O, Min),
                (Min-R)-O @< Key1
            ->  Key2 = (Min-R)-O,
                Pairs1 = [Pair|Pairs2]
            ;   Key2 = Key1,
                Pairs1 = [Pair|Pairs2]
            )
        ),
        mats_labeling(O2, Pairs).

mats_labeling(O, Pairs) :-
        nonvar(O), !,
        mats_labeling(Pairs).
mats_labeling(O, Pairs) :-
        fd_min(O, Min),
        fd_max(O, Max),
        Mid is (Min+Max)>>1,
        (   O #=< Mid,
            mats_labeling(O, Pairs)
        ;   O #> Mid,
            mats_labeling(Pairs)
        ).

post(static, TasksLimits, _, _, _) :-
        (   foreach(Tasks-Limit,TasksLimits)
        do  cumulatives(Tasks, [machine(1,Limit)], [bound(upper)])
        ).
post(uni, TasksLimits, _, _, _) :-
        (   foreach(Tasks-Limit,TasksLimits)
```

```
        do  clpfd:uni_cumulative(Tasks, [limit(Limit)])
        ).
post(decomposed, TasksLimits, _, _, _) :-
        (   foreach(Tasks-Limit,TasksLimits)
        do  (   foreach(task(O,D,E,H,I),Tasks),
                foreach(task(O,D,E,[H],I),MTasks)
            do  true
            ),
            clpfd:multi_cumulative(MTasks, [Limit])
        ).
post(multi, TasksLimits, Hss, _, _) :-
        (   foreach(_-Limit,TasksLimits),
            foreach(Limit,Limits)
        do  true
        ),
        TasksLimits = [Tasks-_|_],
        (   foreach(task(O,D,E,_,I),Tasks),
            foreach(task(O,D,E,Hs,I),MTasks),
            foreach(Hs,Hss)
        do  true
        ),
        clpfd:multi_cumulative(MTasks, Limits).
post(multi_precedences, TasksLimits, Hss, Ps, _) :-
        (   foreach(_-Limit,TasksLimits),
            foreach(Limit,Limits)
        do  true
        ),
        TasksLimits = [Tasks-_|_],
        (   foreach(task(O,D,E,_,I),Tasks),
            foreach(task(O,D,E,Hs,I),MTasks),
            foreach(Hs,Hss)
        do  true
        ),
        clpfd:multi_cumulative(MTasks, Limits, [precedences(Ps)]).

nomulti(static).
nomulti(uni).
nomulti(decomposed).

disjunctives(Tasks, Lim) :-
        (   foreach(Task,Tasks),
            foreach(H-Task,KL1)
        do  Task = task(_,_,_,H,_)
        ),
        keysort(KL1, KL2),
        reverse(KL2, KL3),
        (   fromto(Lim,H1,H2,_),
            fromto(KL3,[H2-Task2|KL4],KL4,_),
            fromto(Disj,Disj1,Disj2,[]),
            fromto(>,_,Cmp,<),
            param(Lim)
        do  (   H1+H2 > Lim
            ->  Disj1 = [Task2|Disj2]
            ;   Disj1 = Disj2
            ),
            (   KL4 = [] -> Cmp = (<)
            ;   H1+H2 =< Lim -> Cmp = (<)
            ;   Cmp = (>)
```

```
            )
        ),
        (   fromto(Disj,[task(S1,_,E1,_,_)|Disj3],Disj3,[])
        do  (   foreach(task(S2,_,E2,_,_),Disj3),
                param(S1,E1)
            do  E1 #=< S2 #\/ E2 #=< S1
            )
        ).

nbjobs(j30, 30).
nbjobs(j60, 60).
nbjobs(j90, 90).
nbjobs(j120, 120).

generate(Dir, Abs, Rel, Algo, Ss, Ds, Es, Hss, Precedences,
         Tasks1-Lim1, Tasks2-Lim2, Tasks3-Lim3, Tasks4-Lim4) :-
        nbjobs(Dir, NJ),
        bounds(Rel, LCT, _),
        see(Abs),
        skip_lines(19),
        (   for(_,1,NJ),
            foreach(Succs,Succss)
        do  read_ints([_,_,_|Succs])
        ),
        skip_lines(6),
        (   for(_,1,NJ),
            foreach(S,Ss),
            foreach(Dur,Ds),
            foreach(Hs,Hss),
            foreach(E,Es),
            fromto(Tasks1,Tasks1a,Tasks1b,[]),
            fromto(Tasks2,Tasks2a,Tasks2b,[]),
            fromto(Tasks3,Tasks3a,Tasks3b,[]),
            fromto(Tasks4,Tasks4a,Tasks4b,[]),
            param(LCT,Algo)
        do  read_ints([_,_,Dur|Hs]),
            Hs = [R1,R2,R3,R4],
            S in 0..LCT,
            E in 0..LCT,
            S + Dur #= E,
            (   R1=:=0, nomulti(Algo) -> Tasks1a = Tasks1b
            ;   Tasks1a = [task(S,Dur,E,R1,1)|Tasks1b]
            ),
            (   R2=:=0, nomulti(Algo) -> Tasks2a = Tasks2b
            ;   Tasks2a = [task(S,Dur,E,R2,1)|Tasks2b]
            ),
            (   R3=:=0, nomulti(Algo) -> Tasks3a = Tasks3b
            ;   Tasks3a = [task(S,Dur,E,R3,1)|Tasks3b]
            ),
            (   R4=:=0, nomulti(Algo) -> Tasks4a = Tasks4b
            ;   Tasks4a = [task(S,Dur,E,R4,1)|Tasks4b]
            )
        ),
        skip_lines(4),
        read_ints([Lim1,Lim2,Lim3,Lim4]),
        seen,
        gen_precedences(Algo, NJ, Succss, Es, Ss, Precedences).
```

```
gen_precedences(multi_precedences, NJ, Succss, _, _, Precedences1) :- !,
        (   count(I1,1,_),
            foreach(Succs1,Succss),
            fromto(Precedences1,Precedences2,Precedences5,[]),
            param(NJ)
        do  (   foreach(J,Succs1),
                fromto(Precedences2,Precedences3,Precedences4,Precedences5),
                param(I1,NJ)
            do  J1 is J-1,
                (   J1 =< NJ -> Precedences3 = [I1-J1|Precedences4]
                ;   Precedences3 = Precedences4
                )
            )
        ).
gen_precedences(_, _, Succss, Es, Ss, []) :-
        (   foreach(Succs1,Succss),
            foreach(Ei,Es),
            foreach(Si,Ss),
            param(Ss)
        do  (   foreach(J,Succs1),
                param(Si,Ei,Ss)
            do  J1 is J-1,
                (nth1(J1, Ss, Sj) -> Ei #=< Sj ; true)
            )
        ).

skip_lines(N) :-
        (   for(_,1,N)
        do  \+ \+ read_line(_)
        ).

read_ints(Ints) :-
        read_line(Line),
        parse_ints(Line, Ints).

parse_ints([], []).
parse_ints([Dig|Line], [Int|Ints]) :-
        Dig >= 0'0, Dig =< 0'9, !,
        Int0 is Dig - 0'0,
        parse_ints(Line, Int0, Int, Ints).
parse_ints([_|Line], Ints) :-
        parse_ints(Line, Ints).

parse_ints([Dig|Line], Int0, Int, Ints) :-
        Dig >= 0'0, Dig =< 0'9, !,
        Int1 is 10*Int0 + Dig - 0'0,
        parse_ints(Line, Int1, Int, Ints).
parse_ints(Line, Int, Int, Ints) :-
        parse_ints(Line, Ints).
```